Flexible Bindungstechniken für ubiquitäre Ressourcen in verteilten Anwendungen

Studienarbeit

vorgelegt beim Fachbereich Informatik
der Universität Hamburg
Arbeitsbereich Datenbanken und Informationssysteme

von
Nastaran Vaziri Pour
Matrikel-Nummer:
4293460

März 1996

Betreuer:
Prof. Dr. Joachim W. Schmidt

Inhaltverzeichnis

1 Einleitung 1

1.1 Motivation und Zielsetzung 2

1.2 Gliederung der Arbeit 4

2 Konzepte und Methoden des Datenaustausches 5

2.1 Lineare Datenrepräsentationen zur Netzwerkkommunikation 6

2.1.1 DCE: ASN.1 6

2.1.2 Sun-RPC: XDR 7

2.1.3 Datenstrombasierter Daten-, Code- und Threadaustausch 7

2.2 Effektive Übertragungstechniken für komplexe Datenstrukturen 8

2.2.1 Transcopying in DPS-algol 9

2.2.2 Netzwerkreferenzen in Obliq und Emerald 10

2.2.3 Flexible Objektmobilität in Emerald 11

2.2.4 Dynamisches Rebinden ubiquitärer Ressourcen in Tycoon 11

2.2.5 Automatische Replikation in Tycoon 13

2.2.6 Methodenvergleich 14

3 Implementierungsumgebung: das Tycoon-System 17

3.1 Die Tycoon-Systemarchitektur 17

3.2 Die TSP-Schnittstelle 19

3.2.1 Datenrepräsentation des TSP 19

3.2.2 Operationen auf Speicherobjekten 22

4 Implementierung der Bindungstechniken für ubiquitäre Ressourcen in Tycoon 23

4.1 Datenaustausch zwischen persistenten Objektspeichern: Parametrisierte Linearisierung von Objektgraphen 24
4.2 Dynamisches Rebinden ubiquitärer Ressourcen 28
4.2.1 Registrierung von Tycoon-Objekten als ubiquitäre Ressourcen 28
4.2.2 Integration des dynamischen Rebindens in den Linearisierungsalgorithmus 32
4.2.3 Verwendung des dynamischen Rebindens in verteilten Anwendungen 34
4.3 Implementierung der automatischen Replikation 35
4.3.1 Integration der automatischen Replikation in den Linearisierungsalgorithmus 36
4.3.2 Verwendung der automatische Replikation in verteilten Umgebung 37
4.3.3 Integration der automatischen Replikation in Tycoons polymorphe Client-Server-Programmierumgebung 37

5 Performanzsteigerung 39

5.1 Wirkung des dynamischen Rebindens bei der Migration von Threads 39
5.2 Automatische Replikation in Client-Server-Anwendungen 40

6 Zusammenfassung und Ausblick 43

Literaturverzeichnis 45

Kapitel 1

Einleitung

Verteilte Anwendungen entwickeln sich hin zu größeren organisatorischen Zusammenhängen und damit zu größerer Komplexität. Deswegen stellen sie neue Anforderungen an verteilte Systeme und Programmierumgebungen, zu deren Lösung neue Perspektiven der Anwendungsentwicklung geschaffen werden müssen.
Als Beispiel für komplexe verteilte Anwendungen kann die Klasse der Workflow[1]-Anwendungen erwähnt werden, die verteilte Geschäftsprozesse unterstützt. In diesem Zusammenhang treten anspruchsvolle Forderungen auf:

- Flexible, ökonomische Programmierung (mit grafischen Werkzeugen und interaktiv).
- Adäquater Einsatz von Mitarbeitern und Ressourcen.
- Flexible Ausnahmebehandlung in jeder Geschäftsprozeßinstanz.
- Prozeß- und Datensicherheit.
- Realisierung der zeitlichen und räumlichen Ausdehnung von Aktivitäten.

Die Realisierungen verteilter Geschäftsprozesse mit konventionellen verteilten Programmierumgebungen können die genannten Forderungen nur bedingt erfüllen. Ein konventionelles Client-Server-Modell hat die disjunkte Aufteilung des Zustandes (Kontextes) des Workflows in Client- und Server- Kontexte zur Folge, was nicht immer günstig und einfach ist. Die Umsetzung der zeitlichen Ausdehnung der Workflow-Aktivitäten bedingt Client-Server-Bindungen, die über Tage, Wochen oder sogar noch länger andauern. Das Aufrechterhalten von Client-Server-Bindungen für so lange Zeiträume ist heute meistens aus technischen Gründen nicht machbar.

Moderne Sprachen wie TL (Tycoon Language) [Matt93] bieten neue Sprachelemente und innovative Ansätze zur Realisierung komplexer verteilter Anwendungen. In Tycoon sind persistente migrierende Threads (leichtgewichtige Prozesse) für die Migration von Aktivitäten in verteilten persistenten Systemen vorhanden. Threads können sich an

[1] Workflow ist ein abgrenzbarer, arbeitsteiliger Geschäftsprozeß oder Vorgang.

Netzwerkressourcen, die im jeweiligen Knoten vorhanden sind, dynamisch binden. Diese Eigenschaft macht sie ideal zur Implementierung der langlebigen verteilten Anwendungen wie z.B. verteilte Geschäftsprozesse. Das Übersenden von Daten, Codes, Threads, Prozessen, Objekten usw., im weiteren Ressourcen genannt, über das Netz, ist eine neuartige Lösung, die modernen Sprachen eine flexible, dynamische Netzwerkprogrammierung ermöglicht. Die Mobilität der Ressourcen bietet folgende Vorteile:

- Ressourcen mit intensiver Wechselwirkung können auf demselben Knoten unterkommen, um die Kommunikationskosten zu reduzieren.
- Die Ressourcen sind überall verfügbar.
- Die besonderen Hard- und Softwarefähigkeiten eines speziellen Knoten können mitbenutzt werden.
- Mobilität der Ressourcen schafft einen einfachen Weg für Anwender, um Daten zwischen Knoten ohne explizites Marshalling zu transformieren.
- Filetransfer-Mechanismen werden nicht mehr benötigt.

Eine unbeschränkte Mobilität der Ressourcen ist ein Wunschbild jeder modernen Sprache. Die Ressourcen einer Programmierumgebung können besondere Eigenschaften besitzen, die ihre Mobilität einschränkt oder sogar unmöglich macht. Eine plattformabhängige Software oder eine lizenzierte Software könnte als Beispiele für immobile Ressourcen aufgezählt werden. Die komplex rekursiven Daten, Standard-Bibliotheksmodule und Datenbanken sind Ressourcen mit sehr großen Strukturen, deren Mobilität aus ökonomischen Grunden (bzgl. Speicherplatz und Übertragungszeit) sehr kritisch werden kann. Die zu übertragenden Datenobjekte in modernen Sprachen setzen sich meistens aus mobilen bzw. immobilen Ressourcen zusammen. Durch sorgfältiger Entwicklung und Optimierung der Kommunikationskomponenten, die die immobilen Ressourcen während der Übertragung unter Berücksichtigung ihrer besonderen Eigenschaften behandeln, kann eine virtuelle Mobilität derartiger Ressourcen erreicht werden.

1.1 Motivation und Zielsetzung

Tycoon[2] ist eine integrierte persistente und polymorphe Programmierumgebung, die im Rahmen des FIDE-Projekts am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg entwickelt und implementiert wird.
Während des letzten Jahres hat die Entwicklung der Kommunikationsmechanismen als Add-on-Komponenten und die Portierung der Tycoon-Sprache (TL) auf verschiedenen Plattformen die Skalierung des Tycoon-Systems zur Folge gehabt. Die persistente Tycoon-Sprache kann als eine mächtige Scripting-Sprache[3] für verteilte Anwendungen eingesetzt werden.[MMS95b]

---

[2] Tycoon steht für Typed Communicating Objects in Open Environments.
[3] Die Scripting-Sprachen sind interpretierte Sprachen wie Perl, Python, WIntegrate, usw.

Die verteilten Idiome in Tycoon erstrecken sich vom polymorphen, entfernten Prozeduraufruf höherer Ordnung bis zur Migration von Threads. Diese Idiome benutzen die auf der Systemebene existierenden Techniken zur Übertragen von Daten, Code, Threads als Basis für die Kommunikation zwischen autonomen Seiten.
Die Deep-Copy-Operation ist die grundlegende Semantik der Datenübertragung (shipping) zwischen den Objektspeichern im Tycoon-System. Diese Operation ermöglicht eine lineare Darstellung des transitiven referenziellen Funktionsabschlußes des persistenten Objektes, der als Objektgraph im Objektspeicher abgebildet ist.
Deep-Copying ist bezüglich Speicherplatz und Übertragungszeit kritisch, und zwar nicht nur für Massendaten, sondern auch für Objekte die Referenzen auf Code in Form von Funktionen oder Threads haben. Im Tycoon-System werden die Standard-Bibliotheken, die grafischen Werkzeuge, die Kommunikations-Software, die Fonts, Tabellen und die Implementation von Massendaten beinhalten, sehr häufig von den Tycoon-Anwendungen benutzt.
Die Tatsache, daß die Standard-Bibliotheken in jedem Tycoon-System vorhanden sind und als ubiquitäre Ressourcen betrachtet werden können, kann für die Reduzierung des Kommunikationsverkehrs genutzt werden.
Die Grundidee ist einfach: warum sollen diese ubiquitären Ressourcen über das Netz geschickt werden? Es wäre weitaus ökonomischer, nur wenige Bytes umfassende, weltweit eindeutige symbolische Referenzen über das Netz zu senden, die vom Empfänger identifiziert und durch entsprechende lokale Ressourcen ersetzt werden müssen. Für die Verwirklichung dieser Idee ist die Existenz der flexiblen Bindungstechniken für ubiquitäre Ressourcen eine wichtige Voraussetzung.

Ziel der vorliegenden Arbeit ist die Implementierung und Integrierung dieser Bindungstechniken in die Tycoon Programmierumgebung. Ausgehend von der erwähnten Idee wird die Methode des dynamischen Rebindens der ubiquitären Ressourcen als eine alternative Methode zur Deep-Copying vorgestellt. Diese ermöglicht die Identifizierung ubiquitärer Ressourcen während der Linearisierung eines Objektgraphen und ersetzt sie durch deren symbolische Referenzen. Es soll dafür gesorgt werden, daß diese symbolischen Referenzen während der Deserialisierung eines Objektgraphen am Zielort wiederum durch die lokal entsprechende Ressourcen ersetzt werden. Die Integration des dynamischen Rebindens in den in Tycoon vorhandenen Linearisierungsalgorithmus soll streng modular erfolgen.

Dynamisches Rebinden kann nicht stattfinden, wenn beim Ankommen eines Symboles kein entsprechendes lokales ubiquitäres Objekt vom Empfänger identifiziert wird. Das System soll nicht nur mit einer Fehlermeldung auf diesen Fall reagieren, sondern bei der Existenz einer bidirektionalen Kommunikationsleitung eine automatische Replikation, die für Benutzer transparent ist, einsetzen. Diese soll für die Übertragung der fehlenden Ressourcen und deren Registrierung als ubiquitär beim Empfänger sorgen.

Diese Bindungstechniken sollen sich gut mit verteilten Programmierungsidiomen im Tycoon System vermischen und zur Performanzsteigerung der Datenübertragung in verteilten Anwendungen führen.

1.2 Gliederung der Arbeit

In Kapitel 2 werden die wichtigsten Methoden und Konzepte der Datenübertragung in der konventionellen bzw. modernen Netzwerkprogrammierung vorgestellt und diskutiert. Des weiteren werden die im Rahmen dieser Arbeit entwickelten Methoden, dynamisches Rebinden und automatische Replikation, konzeptionell dargestellt.

Es folgt in Kapitel 3 eine kurze Darstellung der Tycoon-Architektur. Anhand dieser wird auf das Tycoon-Store-Protocol (TSP) eingegangen. Dieser wesentliche Bestandteil der Tycoon-Implementation beinhaltet Techniken zur Unterstützung von datenstrombasierter Datenübertragung (Migration von Daten, Code und Threads) zwischen Tycoons Objektspeichern.

Kapitel 4 beinhaltet zuerst eine technische Beschreibung der Implementierung der parametrisierten Linearisierung von Objektgraphen, die die grundlegende Technik des Datenaustausches zwischen persistenten Objektspeichern ist. Weiterhin befaßt sich das Kapitel mit der Implementierung und Einbettung der flexiblen Bindungsteckniken d.h. dynamisches Rebinden und automatischer Replikation im Tycoon-System. Dabei wird die modulare Integration dieser Bindungstechniken in den Linearisierungsalgorithmus näher betrachtet. Es folgt die Verwendung dieser Methoden in verteilten Anwendungen.

Die Performanzsteigerung der Datenübertragung in verteilten Anwendungen nach Benutzung des dynamischen Rebindens und automatischer Replikation wird in Kapitel 5 demonstriert.

Die Arbeit schließt mit einer Zusammenfassung der wesentlichen Punkte und einem Ausblick auf mögliche Erweiterungen der hier beschriebenen Implementierung.

Kapitel 2

Konzepte und Methoden des Datenaustausches

Datenaustausch zwischen verschiedenen Netzknoten ist eine elementare Voraussetzung, um verteilte Anwendungen erstellen zu können. Die Abwicklung der Kommunikation, d.h. der Austausch von Daten über ein Medium, wird von unterschiedlichen Produkten als Dienst erbracht. Diese sogenannte Middleware wird von verschiedenen Anbietern wie SUN, BSD oder OSF [Corb91, OSF93] mit teilweise stark unterschiedlichen Philosophien und unterschiedlichen Umfang bereitgestellt und in der Praxis zur Erstellung verteilter Anwendungen eingesetzt.

Eine wichtige Phase während der Kommunikation ist das Codieren und Decodieren der Aufrufe, d.h. die Umwandlung der zu übertragenden Daten in eine Darstellung, die für beide Kommunikationspartner normiert ist. Serialisierung bzw. Deserialisierung der Daten ist ein wesentlicher Bestandteil des Codierens bzw. Decodierens, denn die Übertragungsmöglichkeiten der meisten Kommunikationskomponenten erlauben nur die Übertragung von seriellen Datenströmen. Weiterhin müssen strukturierte Daten, die einen Objektgraph besitzen, linearisiert werden, um die transitive Referenzierung zu behalten. Die Rekonstruktion zum ursprünglichen Objektgraph erfolgt durch die Delinearisierung.

In konventionellen Netzwerkprogrammierumgebungen werden nur Daten, die keinen Programmcode beinhalten, ausgetauscht. Codieren und Linearisieren dieser Daten kann von herkömmlicher Middleware ausgeführt werden. Die modernen Sprachen, die Daten- und Codeaustausch für verteilte Anwendungen zugrunde setzen, benötigen hingegen ergänzende Techniken für Linearisierung und Codierung der zu übertragenden Daten. Die Abbildung (2.1) soll diesen Unterschied darstellen.

Zunächst werden die linearen Datenrepräsentationen in Sun-RPC und DCE-RPC, die in der konventionellen Netzwerkprogrammierung für das Versenden von Daten eingesetzt werden, vorgestellt. Im folgenden werden intelligente Bindungstechniken zur Behandlung von komplexen Datenstrukturen und besonders Programmcode während

der Linearisierung und Delinearisierung in unkonventionellen Netzwerkprogrammier-umgebungen diskutiert. Dabei werden die Methoden dynamisches Rebinden und automatische Replikation, deren Implementierung Schwerpunkt dieser Arbeit ist, ausführlich behandelt.

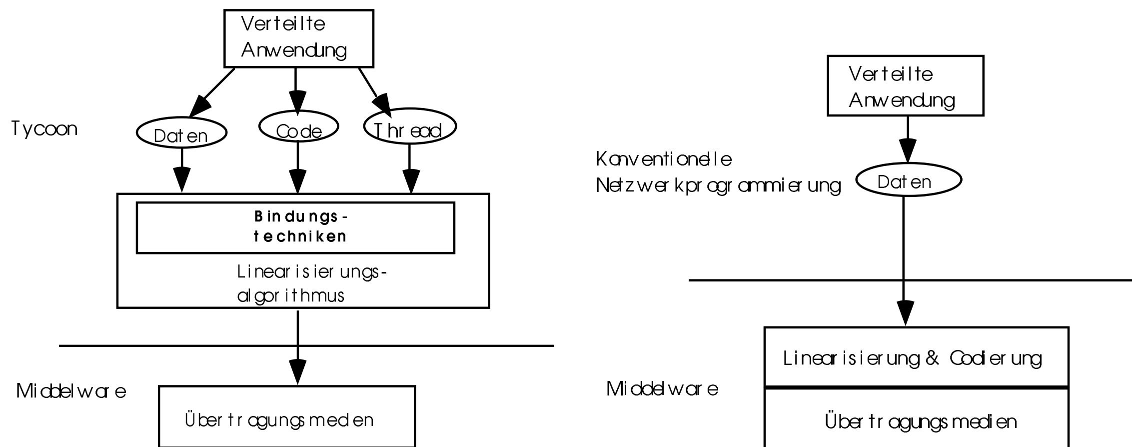


Abbildung 2.1: Datenlinearisierung in verteilten Anwendungen

2.1 Lineare Datenrepräsentationen zur Netzwerkkommunikation

In **heterogenen** Netzen ist der Datenaustausch im allgemeinen mittels einer externen linearen Datenrepräsentation möglich. Dabei sollen die Daten ihre Bedeutung unabhängig von der Architektur der beteiligten Knotenrechner behalten. Die Kommunikations-komponenten der DCE-RPC und Sun-RPC bieten ihren Anwender eine lineare externe Datenrepräsentation der Daten, die keine Programmcode beinhalten. Deep-Copying erlaubt die Linearisierung beliebiger Sprachobjekte.

2.1.1 DCE-RPC

Das Distributed Computing Environment (DCE) der Open Software Foundation (OSF) stellt im Rahmen seiner Unterstützung zur Programmierung von verteilten Systemen den DCE-RPC zur Verfügung. Die zu übertragende Parameterstruktur des DCE-RPC darf von beliebig komplexer Struktur sein. Fast alle Verzeigerungen, die in der Programmiersprache C möglich sind, werden akzeptiert, wobei die referenzierten Daten in den Adreßraum des Kommunikationspartners kopiert werden. Eine nicht erlaubte Verzeigerung stellt z.B. ein Zeiger auf eine Prozedur dar. Zur Übertragung größerer Datenmengen sind Pipes eingeführt. Unter Angabe einer Referenz auf die jeweilige Pipe kann sich der Kommunikationspartner während der Aufrufbearbeitung weitere Daten aus der Pipe anfordern, ohne daß es einer kompletten Übertragung der gesamten Daten zu Beginn des Aufrufes bedarf [Schi 93].

2.1.2 Sun-RPC: XDR

Der Sun-RPC ist eine der am weitesten verbreiteten RPC-Implementierungen. Grundlage des Sun-RPC ist die External Data Representation (XDR) [SUN90], die sowohl für die einheitliche Darstellung des RPC-Protokolls als auch zur Konvertierung der zu übertragenden Daten verwendet wird.
Die External Data Representation (XDR) ist eine kanonische Darstellungsform von Daten. Ihr Zweck ist es, eine einheitliche Datendarstellung zur Verfügung zu stellen, die unabhängig von der jeweiligen lokalen Darstellungsform des Rechnerknotens ist. Die Codierung und Decodierung der Daten erfolgt in bzw. aus Datenströmen, den sogenannten XDR-Streams, die durch den abstrakten Datentyp XDR repräsentiert werden. XDR-Routinen unterstützen sämtliche Datentypen und komplexen Datenstrukturen, die keinen Code erhalten.

2.1.3 Datenstrombasierter Daten-, Code- und Threadaustausch

Viele moderne Sprachen wie Tycoon [Matt93], Amber [Car86], Quest [Car89], und Napierl88 [Mun 93], legen die Methode Deep-Copying, Linearisierung und Schreiben des ganzen transitiven referenziellen Abschlusses (Closure) eines Objektgraphen in einen File, für Datenshipping und -austausch zugrunde.

Die in Tycoon realisierte Deep-Copy-Operation besitzt folgende Vorteile:

- Neben Daten können zusätzlich auch Code sowie Threads linearisiert werden.
- Die Daten können in einen beliebigen Bytestrom kopiert werden (z.B. Betriebssystem-Stream, Netzwerkkommunikationkanäle, usw.)
- Für den Datenaustausch in heterogenen Netzen sind effiziente Konvertierungs-Mechanismen vorhanden.
- Benutzerdefinierte Methoden, die Bindungen zur immobilen Ressourcen behandeln, können installiert werden.

Die Abbildung (2.2) zeigt die Linearisierung des Objektgraphen der Funktion *info* mittels Deep-Copying. Sie wird vom Modul *statistic* exportiert und vom Adreßraum A in den Adreßraum B übertragen werden muß.
Der Objektgraph der Funktion *info* beinhaltet Referenzen auf die importierten Module *database* und *print* und benutzt die Funktionen *print.int* und *database.count*. Während der Linearisierung bleibt diese referenzielle Integrität erhalten. Die Module *print*, *int*, *database* und *count* werden ebenfalls übertragen. Im Adreßraum B wird der Objektgraph der Funktion *info* rekonstruiert.

```
module statistic
import

  database       (* Eine sehr große Datenbank*)
```

```
  print             (* Ein Standard Ausgabemodul*)
export
  ...
  let info() = print.int(database.count())
  ...

end;
```

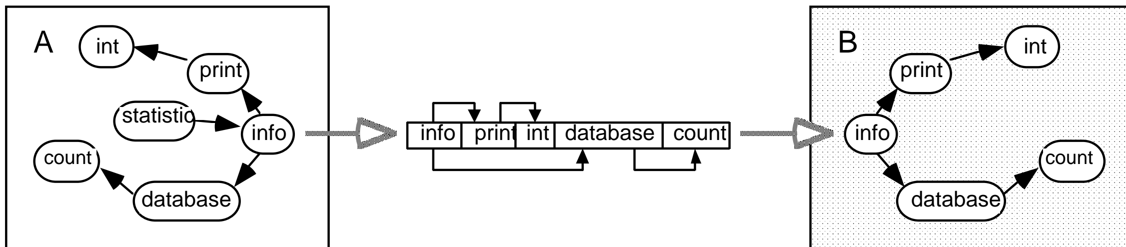


Abbildung 2.2: Deep-Copying eines Objektgraphen

Deep-Copying ist bezüglich Speicherplatz und Übertragungszeit kritisch. Dies ist nicht nur bei der Übertragung von Massendatenstrukturen wie Listen, Bäumen, Datenbanktabellen, sondern auch für Objekte, die Code referieren. Die Funktionen beinhalten mehrere Referenzen auf andere Funktionen, die selbst weiter Referenzen auf Funktionen beinhalten und so weiter. Die Situation wird noch kritischer bei Aggregation der Funktionen in Modulen oder Klassen, wobei die Funktionen, die zum Ausführen nicht benötigt werden, indirekt referenziert werden.

2.2 Effektive Bindungstechniken für komplexe Datenstrukturen

Das Übertragen von komplexen rekursiven Datenstrukturen, die Programmcode beinhalten, ist ein wichtiges Problem bei der Implementierung von moderner Netzwerkprogrammierung. Die objektorientierten verteilten Programmierumgebungen setzen die Objekte als Einheit der Verteilung zugrunde. Die Objekte können andere Objekte referenzieren, die sich in einem Graph von Referenzen befinden.
Ein migrierender Thread kann ebenfalls einen größen Objektgraphen besitzen. Das folgende Beispiel, Funktionsabschluß vom migrierenden Thread *t*, das in der persistenten polymorphen Sprache Tycoon implementiert ist, beinhaltet Referenzen auf importierte Funktionen *cCallback, chekpoint, iter, thread, volatile*,.... und *window*. Bei der Migration von *t* müssen sämtliche importierten Funktionen auch mitgesendet werden. Die Abbildung (2.3) illustriert den dazugehörigen Objektgraphen.

```
Import ...
  cCallback chekpoint iter thread volatile..
  brush button pen color window menu outputDevice...
```

```
let activity(self :thread.T(ok)) =
  begin
    let var width = 100
    let var x = 0

    let paint(...) = ....color ... brush ...
    let m = menu.newPopUp(...)
    installCallbacks(.... w ... m ... paint ...)
    loop
      outputDevice.drawLine(x ...)
        x := {x+1} % width
    end
  end
let t = thread.new(activity)
...
dynamic.extern(t ....)
```

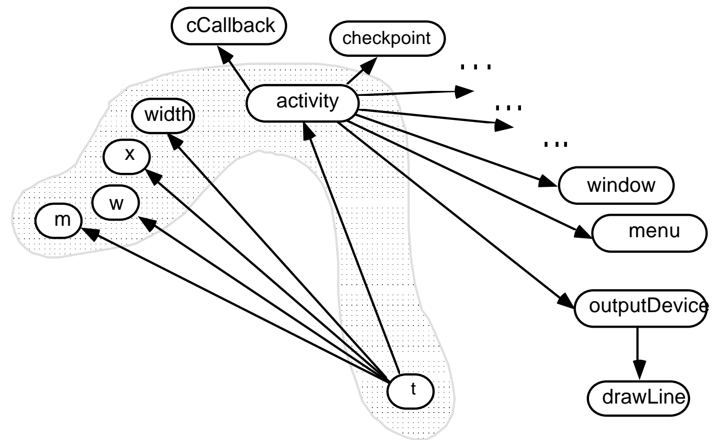


Abbildung 2.3: Der transitiv-referenzierte Objektabschluß

Eine kritische Frage beim Übersenden komplexer Datenobjekte ist: Wie viele Objekte sollen versendet werden? Die zu übertragenden Datenobjekte können sogar immobile Objekte referenzieren. Wie sollen immobile Teile des Objektgraphen behandelt werden? Eine wichtige Einschränkung bei der Übertragung dieser Daten, die rekursive Strukturen besitzen, ist das Erhalten der referenziellen Integrität bzw. das Vermeiden von „dangling Referenzen".

Als eine Lösung kann der Objektgraph ab bestimmten Knoten, die vom Benutzer definiert werden, abgeschnitten werden. So wird der Graph nicht mehr vollständig gesendet. Der weggelassene Subgraph kann durch symbolische Referenz repräsentiert werden. Diese Idee ist der Ansatz der folgenden Methoden und Bindungstechniken.

2.2.1 Transcopying

DPS-algol [Wai88] ist ein von PS-algol abstammendes persistentes System, das die Verteilung durch Einrichtung eines globalen Adreßraumes, der sich auf mehrere Knoten bezieht, unterstützt.
Die Sprache besitzt zwei Konstrukturen *transcopy* und *assign,* womit atomische Daten zwischen verteilten Knoten ausgetauscht werden können. Die Semantik von Transcopying hängt vom Typ des zu kopierenden Objektes ab. Hier wird eine Top-level-Copy vom Objektabschluß dem Deep-Copying bevorzugt. Die Hauptidee ist das Reduzieren der Menge der zu übertragenden Daten und das Verhindern des unachtsamen Kopierens vom ganzen Objektspeicher.
Diese Methode hat die Ausbreitung des Objektgraphen auf mehrere Knoten zur Folge. Die Illusion des globalen Adreßraumes wird durch einheitliche Behandlung der Objektreferenzen erreicht. Diese wird von DPS-algol abstrakten Maschine, die die

Referenzen auf lokale Daten von denen auf entfernte Daten unterscheiden kann, unterstützt. Die Referenzen auf entfernte Objekte werden durch *Remote-Pointer*, die die Objekte beim Benutzen der kontextuellen Adressierung eindeutig identifizieren, implementiert. Die abstrakte Maschine, die auf einem lokalen Store läuft, generiert einen *Remote-Pointer*, wenn sie eine Referenz, der ein entferntes Objekt referiert, im Store entdeckt. Jeder Prozeß behält eine Exporttabelle von *Remote-Pointern*, die er exportiert hat. Somit hängt der Zusammenhang der verteilten Objektspeicher von der Übereinstimmung der Exporttabelle im Store und von der Benutzung der *Remote-Pointer* vom Prozeß ab.

2.2.2 Netzwerkreferenzen in Obliq und Emerald

In Emerald [Jul88] und der von Luca Cardelli 1994 entwickelten Sprache Obliq [Card94], die spezielle Sprachsysteme für die verteilte Programmierung sind, ist die Netzwerkreferenz als Sprachelement integriert.
Ein Objekt innerhalb des Objektgraphen wird durch eine Referenz auf seine Netzwerkadresse in der externen linearen Repräsentation des Objektgraphen ersetzt.
Der Aufruf des Objektes zur Laufzeit im Zielknoten hat eine Netzwerkkommunikation mit der Quelle, wo das Objekt sich befindet, zur Folge (Abbildung 2.4). Netzwerkreferenzen haben die Datenfragmentierung auf verschiedenen Netzknoten zur Folge. Die Prozesse (Threads) sind nicht mehr autonom. Die Methode ist vorallen bei der netzweiten Benutzung der immobilen Ressourcen (Datenbanken, Drucker, etc.) nützlich. Diese kann auch als eine Modellierung von Referenzen auf veränderliche Objekte in jedem System betrachtet werden, denn sie bewahrt die originale gemeinsame Nutzung der Objekte.

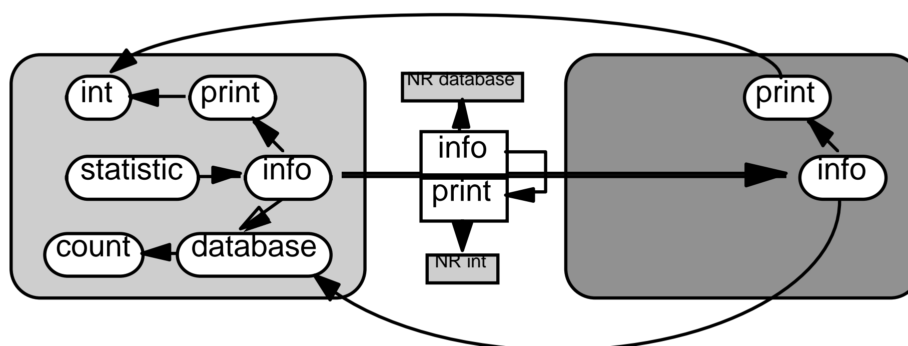


Abbildung 2.4: Konvertierung in Netzwerk-Referenzen

Wegen der Mobilität der Objekte in Emerald ist es möglich, oft vorkommende Netzwerkreferenzen durch das Senden des Argumentobjektes zu vermeiden.
Die entsprechende Entscheidung hängt von:

- der Größe des Argumentobjektes,

- von der Anzahl der Aufrufe,
- von anderen aktuellen oder zukünftigen Aufrufen des Argumentobjektes und
- von den relativen Kosten der Mobilität, Lokalität und entfernten Aufrufes ab.

Dieses Konzept ist in Emerald durch den Parameterübergabemechanismus *call-by-move* realisiert. Die Semantik von *call-by-move* entspricht der Semantik von *call-by-reference*, aber zur Laufzeit wird das Argumentobjekt nicht mehr mit Netzwerkreferenzen referenziert, sondern zum Zielort übertragen.

2.2.3 Flexible Objektmobilität in Emerald

Emerald besitzt keine Klassenhierarchie. Die Objekte sind nicht in Klassen untergebracht, deswegen hat jedes Objekt seinen eigenen Code bei sich. Diese Unterscheidung wird in einer verteilten Umgebung sehr wichtig, wo die Trennung des Objektes von seinem Code von Bedeutung ist. Allerdings benutzen identisch implementierte Objekte auf einem Knoten denselben Code. Bei der Implementierung ist der Code im „Concrete-Type-Object" untergebracht. Die Concrete-Type-Objects sind unveränderlich, deswegen können sie frei kopiert werden. Wenn ein Objekt sich von einem Knoten zum anderen bewegt, werden nur seine Daten übersandt. Falls es einen Prozeß beinhaltet, wird der Prozeßtask auch mit Daten gesendet, aber es wird kein Code übertragen. Wenn der Emerald-Kernel ein Objekt empfängt, kann er bestimmen, ob eine Kopie der Implementierung von Concrete-Type-Object für das ankommende Objekt lokal existiert. Wenn dies nicht den Fall ist, kann Kernel mittels des Location-Algorithmus einen Knoten lokalisieren, der den Code hat, und den Code anfordern.
Normalerweise besitzt der Sender des Objektes auch das Concrete-Type-Object. Wenn das Concrete-Type-Object ankommt, wird es dynamisch in den neuen Kernel eingebunden. Der Compiler führt den dynamischen Bindevorgang durch (Generieren von verlegenden Code, Eintragen in die Informationstabelle). Diese Methode macht es möglich, neue Concrete-Type-Objects für die bereits existierenden abstrakten Typen dynamisch ins Kernelsystem einzufügen. Concrete-Type-Objects bleiben auf einem Knoten, solange sie von anderen Objekten referenziert werden, danach werden sie von Garbage-Collector aufgeräumt.

2.2.4 Dynamisches Rebinden ubiquitärer Ressourcen in Tycoon

Eine Klasse der Ressourcen im Tycoon-System sind ubiquitäre Ressourcen, die in jedem Tycoon-System vorhanden sind. Standard-Bibliotheken, die die Kommunikations-software, grafischen Werkzeuge und die Implementierung der Bulkdaten beinhalten, und Betriebssystemoperationen können als sehr typische Beispiele für ubiquitäre Ressourcen aufgezählt werden. Diese Ressourcen werden sehr häufig von verschiedenen Anwendungen benutzt. Das Migrieren von Threads, die sich auf solche Anwendungen beziehen, hat auch das Übertragen dieser Ressourcen zur Folge. Da diese Ressourcen teilweise sehr groß sind, ist Deep-Copying eine sehr kritische Methode für die richtige Behandlung dieser Ressourcen. Dynamisches

Rebinden der ubiquitären Ressourcen ist als eine Alternative zum Deep-Copying im Tycoon-System im Rahmen dieser Arbeit implementiert worden.
In der Methode dynamisches Rebinden werden die ubiquitären Ressourcen markiert. Zum Zeitpunkt der Übertragung werden die ubiquitären Ressourcen durch symbolische Referenzen, die als global eindeutige Identifier (eg. universal identifier, UID) betrachtet werden können, repräsentiert. Die Referenzen werden vom Empfängersystem identifiziert und durch entsprechende lokale Ressourcen ersetzt.
Die Abbildung (2.5) illustriert die Übertragung der in Tycoon implementierten Funktion *info* zwischen den Adreßräumen A und B mittels dynamischen Rebindens.

```
Module Statistic
import database print;
export
  let info() = print.int(database.count())

end
```

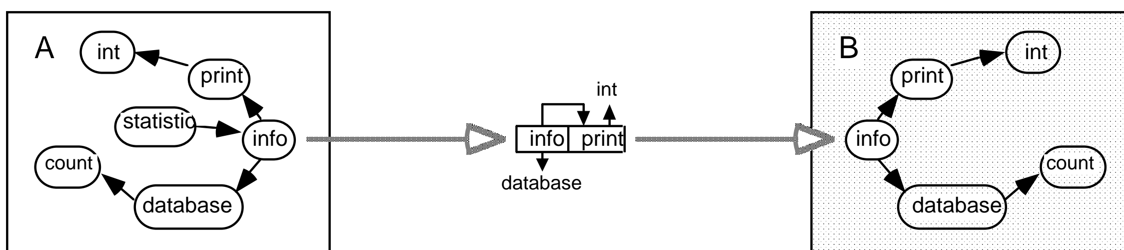


Abbildung 2.5: Dynamisches Rebinden ubiquitärer Ressourcen

Das Modul *Statistic* exportiert die Funktion *info*, die eine Datenbankabfrage beim Aufruf der Funktion *count* im Modul *databas*e hat. Das Ergebnis wird beim Aufruf der Funktion *print.int* auf dem Bildschirm ausgegeben. Die Abbildung (2.5) zeigt den vereinfachten Objektgraphen der Funktion *info* im Adreßraum A. Der Funktionsabschluß von *info* beinhaltet Referenzen auf die Module *database* und *print*. Die Module *print* und *database* befinden sich in den Standard-Bibliotheken vom Tycoon-System und können auf beiden Seiten als ubiquitäre Ressourcen registriert sein. Beim Shippen der Funktion *info* zwischen den Adreßräumen wird nur ein Teil des Moduls *print* und das Modul *info* übersandt. Die Module *database* und *int* werden durch ihre global eindeutigen symbolischen Referenzen im Datenstrom repräsentiert. Im Adreßraum B wird der Objektgraph der Funktion *info* nach der Übertragung wieder komplett aufgebaut. Die symbolischen Referenzen *database* und *int* werden durch lokale Ressourcen ersetzt und in den Objektgraph gebunden.

Dynamisches Rebinden hat eine Reihe von Vorteilen:

- Starke Reduzierung der zu übertragenden Daten und die Minimierung des Netzwerkverkehrs.
- Die Lokalität des Programmes bleibt erhalten.
- Die zu übertragenden Objekte bleiben vollkommen autonom. Dieses ist bei der Migration von Threads besonders wichtig.

Die Einschränkungen bei der Benutzung der Methode sind:

- Der Anwender ist für die Richtigkeit der äquivalenten Semantik von ubiquitären Ressourcen verantwortlich.
- Äquivalente Ressourcen müssen auf beiden Seiten vorhanden sein. Anderenfalls muß die Anwendung der dynamischen Replikation vorgesehen sein.
- Dynamisches Rebinden ist keine allgemeine Alternative für die anderen Bindenmechanismen. Wenn im oben erwähnten Beispiel das Ergebnis eines Aufrufes von *info* von der Seite B aus am Ende auf der Seite A erscheinen muß, dann muß das Modul *print* auf der Seite B durch Netzwerkreferenzen auf das Modul *print* von Seite A repräsentiert werden. Hier kann dynamisches Rebinden nicht eingesetzt werden.

Im Kapitel 4 ist die Implementierung des dynamischen Rebindens im Tycoon-System erörtert.

2.2.5 Automatische Replikation in Tycoon

Dynamisches Rebinden kann nicht stattfinden, wenn beim Ankommen eines Symboles kein entsprechendes lokales ubiquitäres Objekt vom Empfänger identifiziert wird. Dieser Fall kann als Fehlermeldung implementiert werden. Aber bei der Existenz vom Kommunikationskanal kann der Empfänger die unbekannten Symbole zurücksenden und die fehlenden Objekte anfordern. Der Sender übersendet die originalen Ressourcen beim Benutzen der Methode Deep-Copying. Am Zielort werden die ankommenden Ressourcen in den noch nicht vollständigen Objektgraphen integriert und als ubiquitäre Ressourcen registriert. Eine weitere Übertragung der gleichen Ressourcen kann mittels dynamischen Rebindens stattfinden.

In der Abbildung (2.6) migrieren *threads1* und *threads2*. Beide Operationen importieren das Modul *editor*, was als ubiquitäre Ressource im Adreßraum A registriert ist. Beim Migrieren von *thread1* wird das Modul *editor* durch seine symbolische Referenz im Datenstrom repräsentiert. Im Adreßraum B ist das Modul *editor* nicht vorhanden, deswegen kann die symbolische Referenz auf *editor* nicht identifiziert werden. Adreßraum B sendet eine Vermißtenmeldung für Modul *editor* an den Adreßraum A zurück. Adreßraum A übersendet diesmal das Modul *editor* mittels Deep-Copying. Beim Ankommen des Moduls *editor* wird es in den Objektgraphen von *info* integriert und zusätzlich als ubiquitäre Ressource im Adreßraum B registriert. Anschließend migriert *thread2*. Wobei wieder das Modul *editor* durch die symoblische Referenz repräsentiert wird, aber diesmal kann das dynamische Rebinden vollständig

durchgeführt werden, weil das Modul *editor* bereits im Adreßraum B als ubiquitäre Ressource registriert ist, und seine symbolische Referenz kann identifiziert und durch die lokalen Ressource ersetzt werden.

Automatische Replikation hat keine Typprüfung zufolge. Die ubiquitären Objekte sind identisch, weil sie eine gemeinsame Herkunft haben.
Im Abschnitt 4.3 wird die Implementierung der automatischen Replikation ausführlich diskutiert.

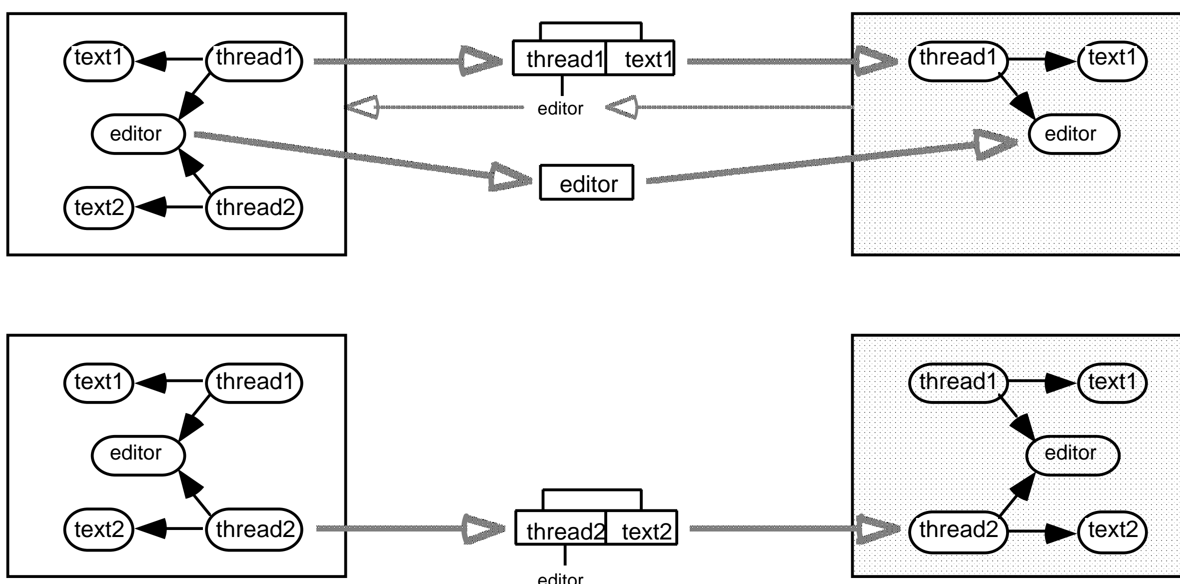


Abbildung 2.6: Die drei Schritte der automatischen Replikation

2.2.6 Vergleich der Methoden zum Übertragen komplexer Daten

Die Minimierung des Kommunikationsaufwandes bezüglich Speicherplatz und Übertragungszeit sowie das Erhalten der Autonomie von migrierenden Threads sind die wichtigsten Kriterien, wonach beim Vergleich zwischen den dargestellten Methoden zum Übertragen von komplexen Daten bewertet wird. Allerdings spielt die Klasse der zu übertragenden Daten eine wichtige Rolle bei der Auswahl der Methode.

Deep-Copying ist für mobile Ressourcen (Dokument, kleiner Code, nicht strukturierte Daten..) sehr geeignet, aber für rekursive komplexe Daten ist diese Methode nicht nur kritisch bezüglich der Kommunikationszeit sondern auch kritisch bezüglich des Speicherplatzes. Die gemeinsame Nutzung der veränderlichen Objekte wird nicht unterstützt. Der Programmier soll diese Ressourcen explizit behandeln.

Netzwerkreferenzen sind für immobile Ressourcen (Datenbanken, Drucker, etc.) sehr wesentlich. Die Umwandlung in entfernte Referenzen hat eine einfache Semantik, denn sie bewahrt die originale gemeinsame Nutzung der veränderlichen Objekte. Andererseits haben die Netzwerkreferenzen Fragmentierung der Codes und vermehrten Übertragungsaufwand beim Zugriff zur Folge. Migrierende Threads, die Netzwerkreferenzen beinhalten, verlieren ihre Autonomie. Ihre Effizienz hängt von hoher Verfügbarkeit der Kommunikationspartner und geringer Netzwerklatenz, wie z.B. LANs, ab.

Dynamisches Rebinden behandelt ubiquitäre Ressourcen (Standard-Bibliotheken, Operationssystemfunktionen, etc.) richtig. Dabei beschränkt sich der Übertragungsaufwand auf den eigentlichen Code und die migrierenden Threads bleiben vollkommen autonom.

Diese Bindungsmechanismen können nebeneinander als Kommunikationsprimitiven existieren. Im Lauf der Übertragung von Daten soll dynamisch entschieden werden, ob ein Deep-Copying durchgeführt oder das Objekt durch Netzwerkreferenz bzw. symbolische Referenzen repräsentiert werden soll.

Kapitel 3

Implementierungsumgebung: das Tycoon-System

Die Implementation der Methoden dynamisches Rebinden und automatische Replikation erfolgte in der streng typisierten persistenten und polymorphen Programmierumgebung Tycoon. Das Tycoon-System ist eine moderne Mehrzweckprogrammierungsumgebung, die besonders für datenintensive Anwendungen geeignet ist. Das System stellt den sprachlichen und architektonischen Rahmen für eine flexible Definition und Integration generischer Dienste in offenen Systemumgebungen bereit. Im Tycoon-System wird der gesamte Prozeß der Integration, Erweiterung, Spezialisierung, Nutzung und Definition generischer Dienste in einem unifizierenden sprachlichen Rahmen abgewickelt. Durch ein portables Laufzeitsystem und die Möglichkeiten für die Netzwerkprogramierung, die über einen Add-On-Ansatz in der Sprache integriert sind, ist Tycoon auch für die Entwicklung von Anwendungen in heterogenen Netzwerken geeignet. Funktionen, Typen und Threads sind Objekte erster Klasse und können daher über das Netz verschickt werden.

Das Gesamtsystem ist in mehrere funktionale Schichten aufgeteilt. Jede dieser Schichten ist für bestimmte Aufgabenbereiche optimiert. Anhand der Tycoon-Systemarchitektur wird das Tycoon-Store-Protocol (TSP), das eine wesentliche Grundlage der Tycoon-Implementation ist und Techniken zur Unterstützung von datenstrombasierter Datenübertragung (Migration von Daten, Codes und Threads) zwischen Tycoons Objektspeichern beinhaltet, näher diskutiert.

3.1 Die Tycoon-Systemarchitektur

Die Tycoon-Systemarchitektur strebt eine konzeptionelle und systemtechnische Trennung von Datenmodellierung (TL), Datenmanipulation (TML) und Datenspeicherung (TSP) an. Diese Trennung leistet einen wichtigen Beitrag zur Portabilität und Skalierbarkeit des Tycoon-Systems.[Matt93]
Die Abbildung (3.1) zeigt das globale Zusammenspiel der die Systemschnittstellen verbindenden Architekturkomponenten.

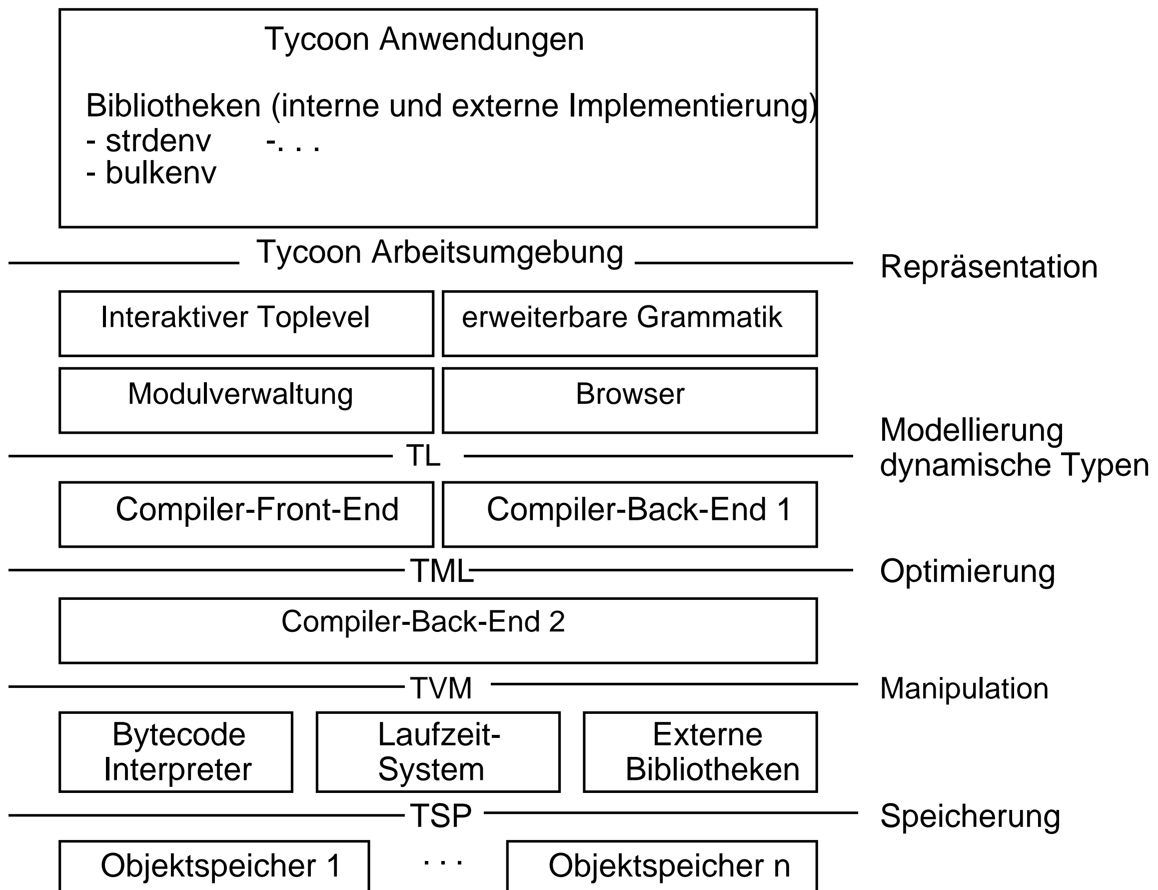


Abbildung 3.1: Die Tycoon-Systemarchitektur

Zur Definition neuer und zur Einbindung bestehender externer Dienste wird die algorithmisch vollständige, strikt typisierte und polymorphe Programmiersprache höherer Ordnung TL verwendet. Sie ermöglicht eine flexible Benennung, Bindung und Typisierung der für datenintensive Anwendungen relevanten Objekte und Dienste. TL bietet strukturell definierte Subtypisierungsregeln für alle Typkonstruktoren. TL wird im Tycoon-System nicht nur zur Datenmodellierung und Applikationsprogrammierung eingesetzt, sondern bildet auch die Systemprogrammiersprache, in der die frei erweiterbaren Tycoon-Bibliotheken und die Tycoon-Sprachprozessoren implementiert sind.
Über die Sprache TML (Tycoon Machine Language), die die Evaluationssemantik einer untypisierten Zwischensprache hat, werden die Daten und die Programmrepräsentation definiert. Sie unterstützt neben einer statischen Zielkodegenerierung insbesondere Portabilität, Interoperabilität in heterogenen Umgebungen und weitreichende dynamische Optimierung analog zur Anfrageoptimierung in Datenbanksystemen
Der Tycoon Byte-Code wird durch die TVM (Tycoon Virtual Machine) ausgeführt. Dieser virtuelle Computer bildet Operationen in Tycoon auf die jeweilige Hardware-Plattform ab.
Das Tycoon-System ist in der Lage, unterschiedliche Objektspeichersysteme transparent zu verwenden. Diese Eigenschaft wird über ein spezielles Protokoll, dem TSP (Tycoon store Protocol). Es definiert eine minimale, weitgehend modellunabhängige Objektspeicherschnittstelle innerhalb der Tycoon Schichtenarchitektur, über das Tycoon alle Zugriffe auf das angeschlossene Objektspeichersystem abwickelt. Im nächsten Abschnitt wird diese Schnittstelle dargestellt.

3.2 Die TSP-Schnittstelle

Das TSP [MMS95c] spezifiziert eine wohldefinierte Schnittstelle zwischen Tycoon-Sprachen (Front-End) und einem frei gewählten Objektspeicher (Back-End). Derzeit werden zwei Tycoon-eigene Stores (*Tymem* und *Tysin*), NapierStor und ObjektStor unterstützt. Das TSP bietet die Abstraktion eines homogenen unbegrenzten Objektspeichers, der das Anlegen, Lesen und Ändern von polymorphen, inhomogenen Datenobjekten gestattet. Diese stabile Programmierungsschnittstelle wird durch eine plattformunabhängige externe Datenrepräsentation (TXR) ergänzt. Die TXR ermöglicht den Austausch von beliebigen, rekursiv strukturierten, persistenten Datenobjekten zwischen TSP-Objektspeichern.

3.2.1 Datenrepräsentation des TSP

Alle semantischen Objekte (Tuple, Mengen, Funktionen, dynamische Typbeschreibung, Module, etc.) der Tycoon-Sprachen werden durch kanonische Abbildung auf primitive Objektspeicherstrukturen im Objektspeicher realisiert. Der Einsatz eines Objektspeichers ermöglicht die Persistenz dieser Tycoon-Objekte. Diese Strukturen können ausschließlich durch Benutzung der wohldefinierten Softwareschnittstelle TSP

manipuliert werden. Jedes im Objektspeicher angelegte Objekt ist eindeutig über eine abstrakte Objektspeicherreferenz identifizierbar.

Elementare Datentypen in TSP

Der Hauptaspekt für die Definition der unterschiedlichen Datentypen in TSP ist die Klassifikation der möglichen Eingabeparameter für die Operationen der Objektspeicherschnittstelle gewesen. Die Beziehungen zwischen den einzelnen TSP-Typen stellt die Abbildung (3.2) dar.

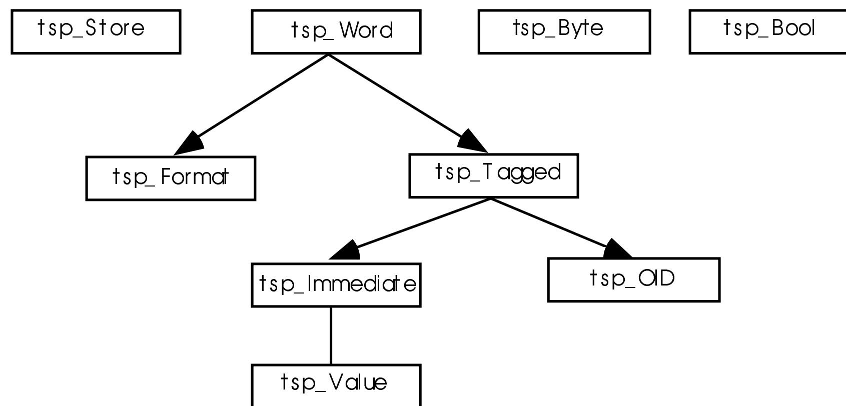


Abbildung 3.2: Elementare Datentypen in TSP

Der Typ *tsp_Store* wird durch den C-Datentyp *void** implementiert. Werte dieses Typs identifizieren eine Objektspeicher.
Der Typ *tsp_Word* ist der Supertyp von den Typen *tsp_Tagged* und *tsp_Format*. Er wird durch den C-Datentyp unsigned int implementiert.
Ein Wert des Typs *tsp_OID* identifiziert ein Objekt eines Objektspeichersystems.
Ein Wert vom Typ *tsp_Immediate* repräsentiert einen atomaren Wert des Benutzersystems.
Der Typ *tsp_Tagged* ist der Supertyp der Typen *tsp_Immediate* und *tsp_OID*. Er wird an den Stellen benutzt, wo die Werte von Typen *tsp_OID* und *tsp_Immediate* gültig sind. Markierungsschemata (tagging) zur Unterscheidung der Objektidentifikatoren von direkten Werten zur Laufzeit sind vorgesehen.
Der Typ *tsp_Byte* wird durch den C-Datentyp unsigned char implementiert.
Der Typ *tsp_Format* wird für die Realisierung unterschiedlicher Objektformate benötigt.
Die Werte vom Typ *tsp_Value* können vom Benutzer manipuliert werden.

Die allgemeinen Datenstrukturen
Jedes Objekt besitzt einen eindeutigen Objektidentifikator, der einen Zugriff auf das Objekt ermöglicht. Die Objektidentifikatoren können in einem Array gespeichert werden, um die Beziehung zwischen Objekten zu modellieren.
Alle Objektarten besitzen die in der Abbildung (3.3) dargestellte Struktur.

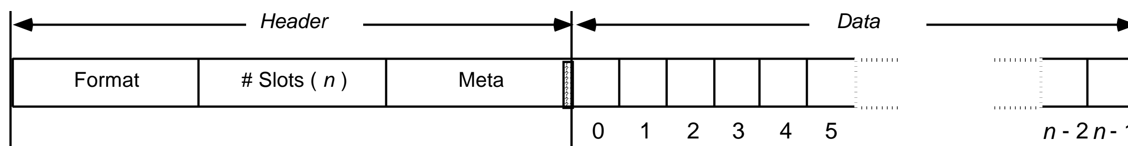


Abbildung 3.3: Allgemeine Objektstruktur in TSP

Ein Objekt besteht aus einem Objektkopf und einer benutzerdefinierbaren Anzahl von Slots.

Objektkopf: Anzahl der Slots, Format des Objektes, Verwaltungsdaten und Meta-Slot sind die Daten, die sich im Objektkopf befinden.
Die Anzahl der Slots kann mindestens Null und maximal unendlich sein, aber es kann bei der Implementierung von TSP eine endliche Zahl als Maximum definiert werden. Der Formatwert wird durch einen vier Byte langen Vektor repräsentiert. Bits 16 bis 31 sind von TSP für interne Informationen belegt. TSP bietet Operationen zum Zugriff auf den Formatwerte eines Objektes. Die Funktion *tsp_getFormat* liefert einen Wert vom Typ *tsp_Word* (32 Bits), bei dem die Bits 16 bis 30 auf null gesetzt sind. Anhand von Bit null und eins kann die Art des Objektes festgestellt werden.

Objektdaten: In Abhängigkeit vom Objektformat können in den Slots Werte vom Typ tsp_Word, tsp-Tagged oder tsp-Byte gespeichert werden.

Klassifizierung von Objekten

Es werden zwei Arten von persistenten Objekte unterstützt: Array-Objekt und Byte-Array-Objekt. Die Art eines Objektes kann auch durch geeignete Formatinformation beim Erzeugen eines Objektes vom Benutzer definiert werden.

Array-Objekt: TSP unterstützt zwei eingebaute Formattypen für Array-Objekte. Das Format bestimmt, welche Werte in Slots des Array-Objektes gespeichert werden können.
Array-Objekte mit dem Format *tsp_Format-TAGGED* enthalten Werte vom Typ *tsp_Tagged*. Die Werte, die in Slots von Arrays gespeichert sind, können vom Type *tsp_Immediate* (OK, Boolean, Charakter, Integer, ...), oder *tsp_OID* sein. Die Größe des Slots ist vier Worte. Um die Objektidentifikatoren von direkten Werten zu unterscheiden, wird das niederwertigste Bit markiert.

Array-Objekte mit dem Format *tsp_Format-Word* beinhalten Werte vom Typ *tsp_Word*. In diesen Arrays ist es nicht möglich, markierte (getaggte) Werte zu speichern.

Byte-Array-Objekt:

Die Größe von einzelnen Slots in Byte-Array-Objekten ist bei der Größe von Werten vom Typ tsp_Byte gegeben. Das vordefinierte Format für Byte-Array-Objekte ist *tsp_Format-Byte*.

3.2.2 Operationen auf Speicherobjekten

Das TSP bietet eine Menge Operationen zur Manipulation von Objekten. Dabei unterscheidet man zwischen Operationen zur Erzeugung von Objekten, Operationen mit denen die Eigenschaften der Objekte verändert oder ermittelt werden können, Operationen, die Daten der Objekte ändern und Operationen zum Export bzw. Import der Objektgraphen aus bzw. in den Objektspeicher.

Operationen zur Erzeugung der Objekte

TSP stellt Operationen *tsp_NewArray* und tsp_NewArrayInt zum Erzeugung von initialisierten bzw. nichtinitialisierten Array-Objekte zur Verfügen. Die Größe, das Format und der Initialisierungswert des Objektes kann vom Benutzer bestimmt werden.

Operationen zur Manipulation der Objektdaten

Ermittelung und Modifikation eines bzw. mehrerer Objekt-Slotwerte geschieht durch die Operationen *tsp_getWord* bzw. *tsp_setWord* und *tsp_getWords* bzw. *tsp_setWords*. Ebenfalls können Daten zwischen Objekten mit *tsp_moveWords* und *tsp_moveBytes* ausgetauscht werden.

Operationen zum Ändern und Abfragen von Eigenschaften der Objekte

Das Format der Objektdaten kann durch die Funktionen *tsp_getFormat* und *tsp_setFormat* ermittelt und geändert werden. Der Zustand des Objektes kann mit *tsp_isImmutable* und *tsp_setImmutable* als veränderlich gesetzt oder abgefragt werden.

Operationen zum Export bzw. Import der Objektgraphen

Die Funktion *tsp_extern* erstellt eine lineare Repräsentation eines beliebigen Objektgraphen zum Export aus dem Objektspeicher. Analog dazu importiert *tsp_intern* den Datenstrom in den Objektspeicher und rekonstruiert den entsprechenden Objektgraph im Objektspeicher. Der Algorithmus dieser Operationen wird in Kapitel 4 ausführlich diskutiert.

Kapitel 4

Implementierung der Bindungstechniken für ubiquitäre Ressourcen in Tycoon

Die Kommunikationsmechanismen von Tycoon sind über Add-On-Ansätze in die Sprache TL integriert. Die verteilte Programmierung im Tycoon-System ist stark von den modernen Eigenschaften der Sprache beeinflußt, sowohl die Funktionen als auch Typen werden als Objekte erster Klasse behandelt. Dies beinhaltet auch das Senden von Funktionen und abstrakten Datentypen über ein Netz. Dieses erlaubt dem Tycoon-Programmierer den Entwurf mächtiger Konstrukte für verteilten Anwendungen. Was widerum den Einsatz der flexiblen Kommunikationskomponenten, wie z.B. die in dieser Arbeit realisierten Bindungstechniken, bedingt.

In diesem Kapitel wird die Implementierung und Einbettung des dynamischen Rebindens und der automatischen Replikation, deren Konzepte in Kapitel 2 vorgestellt wurden, im Tycoon-System beschrieben. Diese Bindungstechniken sind die Systemunterstützung zur Handhabung der Bindungen zu ubiquitären Ressourcen in verteilten Anwendungen. Diese Ressourcen befinden sich wie alle andere Sprachobjekte im Tycoon-System in einem persistenten Objektspeicher und können unter Benutzung des Tycoon-Store-Protocol (TSP) manipuliert werden. Operationen für die Linearisierung der Datenobjekte befinden sich ebenfalls im TSP. Daher ist das TSP als Ausgangspunkt für die Implementierungsphase dieser Arbeit von Interesse.

Um die ubiquitären Ressourcen zu identifizieren, wird die allgemeine Objektstruktur im Objektspeicher erweitert und modifiziert. Eine modulare Integration der Bindungstechniken in den Datenlinearisierungsalgorithmus zieht die Definition neuer Funktionen und damit die Erweiterung des TSP nach sich.

Zunächst wird kurz auf die in TSP vorgesehenen Operationen für die Linearisierung bzw. Delinearisierung der Objektgraphen eingegangen, um darauf aufbauend die Implementierung und Integration der neuen Bindungstechniken, dynamisches Rebinden und automatische Replikation, erläutern zu können.

4.1 Datenaustausch zwischen persistenten Objektspeichern: Parametrisierte Linearisierung von Objektgraphen

Alle Datenobjekte im Tycoon-System wie Werte, Programmcode und Threads werden persistent in einem Objektspeicher gehalten. Diese Objekte können durch Benutzung der wohldefinierten Softwareschnittstelle TSP manipuliert werden.
Im TSP sind Operationen für die Erstellung einer plattformunabhängigen externen Datenrepräsentation (TXR) [MMS95c] von Objektstrukturen vorgesehen. TXR ermöglicht die Übertragung von beliebigen Objekten mittels einer linearen Datenrepräsentation zwischen TSP-Objektspeichern. Diese externe Datenrepräsentation unterstützt zusätzlich die Backup-Operationen.
Die parametrisierten Funktionen *tsp_extern* und *tsp_intern* realisieren die Linearisierung bzw. Rekonstruktion des Objektgraphen. In der vorherigen Version von TSP waren die für *tsp_intern* bzw. *tsp_extern* relevanten Operationen spezifisch für Deep-Copying konzipiert und definiert. Im Rahmen dieser Arbeit, die eine modulare Integration der neuen Bindungstechniken in *intern- /extern*-Algorithmus fordert, sind Änderungen unter Berücksichtigung der folgenden Punkte vorgenommen:

- Die Operationen *tsp_extern* und *tsp_intern* sollen streng modular sein. Die vom Benutzer definierten *writeHandler* bzw. *readHandler* realisieren die Art und Weise der externen Repräsentation einzelner Objekte bzw. deren Einlesen und Erzeugen.

- Der Benutzer soll explizit entscheiden können, welche Objekte miteinander übersendet werden müssen, was bedeutet den Objektgraphen ab einer bestimmten Stelle abschneiden zu können.

- Die Codierung der Daten soll von TSP vollgezogen werden, damit die Datenportabilität garantiert werden kann.

- Der Benutzer soll die Möglichkeit haben, den Objektgraphen der importierten Daten zu traversieren,um intern relevante Operationen darauf ausführen zu können.

Der neue *intern-/extern*-Algorithmus ermöglicht eine flexiblere Linearisierung bzw. Delinarisierung des Objektgraphen. Deep-Copying ist noch als Basis-Konzept für die Linearisierung des Objektgraphen vorhanden. Durch die strenge Modularisierung des Algorithmus ist nicht nur die Integration der in dieser Arbeit zum Ziel gesetzten Bindungstechniken auf eine elegante Weise möglich, sondern sie weist auch den Weg für die Integration anderer Methoden (z.B. Netzwerkreferenzen, s. Kapitel 6).

Im folgenden sollen die *inter-/extern*-Operationen und die in deren Algorithmen beteiligten Funktionen kurz vorgestellt werden:

Die Operation *tsp_extern* akzeptiert als Eingabewert ein beliebiges Objekt im persistenten Objektspeicher und liefert die externe lineare Repräsentation des Objektgraphen zurück.

```
void tsp_extern(tsp_Store store, tsp_Tagged tagged, void *handle,
          tsp_WriteFunction write, tsp_PruneHandler pruneHandler,
tsp_WriteHandler writeHandler)
```

Die Operation *tsp_intern* ist das Pendant zu *tsp_extern*, d.h. sie transformiert eine externe lineare Datenrepräsentation in den entsprechenden Objektgraphen und integriert diesen in den persistenten Objektspeicher. Die Funktion *tsp_intern* konvertiert die externe Repräsentation der Daten automatisch entsprechend die aktuelle Hardware.

```
tsp_Tagged tsp_intern (tsp_Store store, void *handle, tsp_ReadFunction read,
                    tsp_PruneHandler pruneHandler, tsp_ReadHandler
               readHandler, tsp_UpdateHandler updateHandler,
          tsp_SlotHandler slotHandler, tsp-OID *oidTable);
```

Die Funktionen *write* vom Type *tsp_WriteFunction* und *read* vom Typ *tsp_ReadFunction*, die als Funktionsargument in den *intern- / extern_* Algorithmus gegeben sind, kapseln die Datenstromfunktionalität vom *intern- / extern-Algorithmus* ab.

```
typedef tsp_Word(*tsp_WriteFunction)(void *handle, tsp_Byte *pbBuffer, tsp_Word
            nBytes);
```

Eine Funktion vom Typ *tsp_WriteFunction* hat im Prinzip die gleiche Aufgabe wie eine *write* Funktion in der IO-Bibliothek der Programmiersprache C. Sie schreibt *nBytes* aus dem *pbBuffer* in einen Bytestrom, der durch *handle* identifiziert ist. Die Funktion *write* gibt die Anzahl der erfolgreich geschriebenen Bytes zurück.

```
typedef tsp_Word(*tsp_ReadFunction)(void *handle, tsp_Byte *pbBuffer, tsp_Word
            nBytes);
```

Die Funktionalität einer Funktion vom Typ *tsp_ReadFunction* entspricht der von *read* Funktion in Unix, die für einen File definiert ist. Sie liest *nBytes* aus dem Bytestrom, der durch *handle* identifiziert ist, in den *pbBuffer*. Die *tsp_ReadFunction* gibt die Anzahl der Bytes, die vollständig gelesen worden sind, zurück.

```
tsp_Word tsp_externAttributes(tsp_Store store, tsp_XDR xdr, tsp_Tagged
                         taggedMeta, tsp_Format format, tsp_Bool fImmutable,
                    tsp_Word nSlots, void *handle, tsp_WriteFunctions write,
               void *data);
```

Im Objektkopf eines Objektes im Objektspeicher befinden sich Informationen, die die Art und Weise der Linearisierung im *extern*-Algorithmus beeinflussen können. Dazu

gehören Metadaten, Anzahl der Slots, usw. Die Funktion *externAttributes* kapselt Linearisierung des Objektkopfes vom *extern*-Algorithmus ab.

Die folgenden TSP-Typen und TSP-Funktionen ermöglichen dem Benutzer vom TSP, eigene Methode zur Linearisierung eigener atomischer Datenwerten zu realisieren:

```
typedef tsp_Word(*tsp_WriteHandler)(tsp_Store s, tsp_XDR xdr,
                    tsp_OID oid, void *handle, tsp_WriteFunction write);
```

Eine Funktion vom Typ *tsp_WriteHandler* kann vom Benutzer für die Gestaltung der externen Repräsentation einzelner Objekte implementiert werden. Diese ermöglicht z.B. die Integration der Bindungstechniken in den *extern*-Algorithmus (Siehe Anschnitt 4.2.2).

```
void tsp_setWriteHandler(tsp_Store s, tsp_WriteHandler WriteHandler);
```

Der im TSP definierte *WriteHandler* wird durch die vom Benutzer definierte Funktion *WriteHandler* ersetzt.

Die funktion `tsp_WriteHandler tsp_getWriteHandler(tsp_Store s)`

stellt die im TSP definierte Funktion *WriteHandler* zur Verfügung.

```
typedef tsp_OID(*tsp_ReadHandler)(tsp_Store s, tsp_XDR xdr,
                  tsp_Tagged taggedMeta, tsp_Format format,
                  tsp_Bool fImmutable, tsp_Word nSlots,
                  void *handle, tsp_ReadFunction read)
```

Eine Funktion vom Typ *tsp_ReadHandler* kann vom Benutzer zum Einlesen und Erzeugen einzelner Objekte Realisiert werden. Zum Beispiel werden auf diese Weise bei dynamischen Relinken symbolische Referenzen durch lokale Ressourcen ersetzt.

```
void tsp_setReadHandler(tsp_Store s, tsp_ReadHandler ReadHandler);
```

Die Standardfunktion *ReadHandler* im TSP wird durch die vom Benutzer realisierte Funktion *ReadHandler* ersetzt oder zum ersten Mal definiert.

Die Funktion `tsp_ReadHandler tsp_getReadHandler(tsp_Store s);`

Liefert die im TSP aktuell vorhandene Funktion *ReadHandler.*

```
typdef tsp_Bool(*tsp_PruneHandler)(tsp_Store, tsp_OID oid);
```

Eine Operation vom Typ *tsp_PruneHandler* bietet dem Benutzer die Möglichkeit des Abschneidens des Objektgraphen an beliebiger Stelle oder der Zusammensetzung mehrerer Teilgraphen an. Der Benutzer kann explizit spezifizieren, welche Objekte herausgeschrieben und übersendet werden dürfen.

```
typedef void(*tsp_UpdateHandler)(tsp_Store store, tsp_OID oid)
typedef void (*tsp_SlotHandler)(tsp_Store store, tsp_OID oidParent,
                                tsp_Word iIndexParent, tsp_OID oidChild);
```

Schließlich kann mit Hilfe von Funktionen vom Typ *tsp_UpdateHandler* und *tsp_SlotHandler* ein vom Benutzer definiertes Verfahren auf jeden von der *intern* Funktion neu rekonstruierten Objektgraphen ausgeführt werden. Der *UpdateHandler* wird genau für jeden Knoten, d.h. einmal pro Objekt, im Objektgraphen aufgerufen. Die Funktion *SlotHandler* wird einmal pro Kante im übertragenen Graphen aufgerufen. Durch Änderung der vorgefundenen Objektreferenzen können bestimmte Objekte im Graphen komplett durch andere ausgetauscht werden. Zum Beispiel bei automatischer Replikation (siehe Abschnitt 4.3.2) symbolische Referenzen durch nachträglich übertragene Objekte ersetzt.

Deep-Copy-Operation ist in *tymem,* die eine Realisierung des TSP für Tycoon ist, durch die Standardfunktion *defaultwriteHandler* implementiert. Diese Funktion wird als Standardfunktion für die Linearisierung der Tycoon-Objekte vom *extern*-Algorithmus aufgerufen.

```
tsp_word tsp_defaultWriteHandler(tsp_Store store, tsp_XDR xdr, tsp_OID oid,
                                 void *handle, tsp_WriteFunction write, void *data)
```

In *tymem* ist die Standardfunktion *defaultReadHandler* für das Erzeugen eines Objektes, das mit Deep-Copy-Operation linearisiert wurde, vorhanden. Diese Funktion wird vom *intern*-Algorithmus.aufgerufen.

```
tsp_oid tsp_defaultReadHandler(tsp_Store store, tsp_XDR xdr, tsp_Tagged
                          taggedMeta, tsp_word nSlots, void *handle,
                    tsp_ReadFunction read);
```

In dieser Stelle können auf Basis der oben vorgestellten Funktionen die *extern* und *intern* Algorithmen kurz zusammengefaßt werden.

extern-Algorithmus

Die Linearisierung mit Hilfe der *tsp_extern* erfolgt in zwei Phasen. In der ersten Phase wird die transitive Hülle des zu übertragenden Objektes in einer Hash-Tabelle registriert. In der zweiten Phase werden dann die in der Hash-Tabelle eingetragende Objekte sequentiell herausgeschrieben. In dieser Phase wird die Funktion *writeHandler* pro Objekt einmal gerufen. Anhand des vorliegenden Objektformates wird für den Aufruf der entsprechenden *writeHandler*, z.B. *writeHandler* der dynamischen Rebinden für ubiquitäre Ressourcen, entschieden. Allerdings werden zuerst die Attribute des Objektkopfes in kanonischer Weise mit Hilfe der vordefinierten Funktion *tsp_externAttributes* ausgegeben. Die Datenstromfunktionalität ist die Aufgabe der Funktion *write,* die die Daten in den Bytestrom ausgibt. Zusätzlich bietet die Funktion

pruneHandler dem Benutzer die Möglichkeit des Abschneidens des Objektgraphen an beliebiger Stelle oder der Zusammensetzung mehrerer Teilgraphen an.

***intern*-Algorithmus**

Die Delinearisierung [Math96] mittels *tsp_intern* erfolgt ebenfalls in zwei Phasen. In der ersten Phase werden alle Objekte eines zu verarbeitenden Datenstroms sequentiell eingelesen. Im Zuge dessen wird ggf. die Byte-Reihenfolge der aktuellen Hardware angepaßt. Alle neu entstehenden Objektreferenzen werden in einer Tabelle erfaßt, wobei die Position eines Objekteintrages in der Tabelle jeweils dem während der Linearisierung vergebenen Index entspricht. In der zweiten Phase wird sequentiell über die Tabelle iteriert, um alle in Form von Indizies in den Objektinhalten auftretenden Objektreferenzen zu relozieren. Die Funktion *readHandler* ist für das Einlesen und Erzeugen einzelner Objekte zuständig. Sie wird nach dem Einlesen der Attribute des Objektkopfes, die zuerst ankommen, aufgerufen. Anhand dieser Informationen wird entschieden, wie der zu lesende Objektinhalt zu interpretieren ist. Zum Beispiel symbolische Referenzen sollen durch lokale Ressourcen ersetzt werden. Die Funktion *read* besitzt die Aufgabe der Datenstromfunktionalität beim Einlesen. Schließlich kann mit Hilfe von Funktionen *updateHandler* und *slotHandler* ein vom Benutzer definiertes Verfahren auf jeden von der intern neu rekonstruierten Objektgraphen ausgeführt werden. Die Funktion *updateHandler* wird einmal pro Objekt aufgerufen, während *slotHandler* wird einmal pro Kante aufgerufen.

4.2 Dynamisches Rebinden ubiquitärer Ressourcen

Im Kapitel 2 ist bereits dynamisches Rebinden ubiquitärer Ressourcen konzeptuell vorgestellt und beschrieben worden. In diesem Abschnitt wird dessen Implementierung im Tycoon-System gezeigt. Die Realisierung besteht im wesentlichen aus zwei Teilen:

- Registrierung von Tycoon-Objekten als ubiquitäre Ressourcen.
- Integration des dynamischen Rebindens der ubiquitären Ressourcen in den Tycoon-Linearisierungsalgorithmus.

4.2.1 Registerierung von Tycoon-Objekten als ubiquitäre Ressourcen

Dynamisches Rebinden ist orthogonal. Es kann auf beliebige Objekte im Tycoon-System angewandet werden. Ein Objekt wird in folgenden Schritten für das Rebind-Verfahren registriert:

- Erzeugen eines global eindeutigen Symbols,
- Zuordnung vom Objekt zum Symbol und umgekehrt,
- Markierung des Objektes als ubiquitäre Ressource.

Tycoon-Meta-Daten

Um die Implementierung dieser Zuordnung möglich zu machen, wurde im Rahmen dieser Arbeit der Objektkopf der Objektdarstellung im Objektspeicher (siehe Kapital 3) um einen Slot, im weiteren Meta-Slot genannt, erweitert. Dieser Slot ist markiert und kann *Object_Identifier* oder Werte vom Typ *tsp_immediate* beinhalten. Wie es in der Abbildung (4.1) dargestellt ist, kann der Meta-Slot von folgenden Werten belegt werden:

Owner: Er soll zur Identifizierung des Besitzers des Objektes dienen und dabei fremde Zugriffe (Schreiben und Lesen) auf das Objekt vermeiden oder unter Kontrolle bringen.
Equality-Hash-Value/Identity-Hash-Value: Diese Werte werden von sehr speziellen Hashfunktionen generiert. Diese Werte sind beim Verwalten der Objekte mittels der Hashtabellen nützlich.

Referenz: Sie beinhaltet einen Objektidentifikator und wird beim Aufspüren des Symbols vom Objekt und umgekehrt gebraucht.

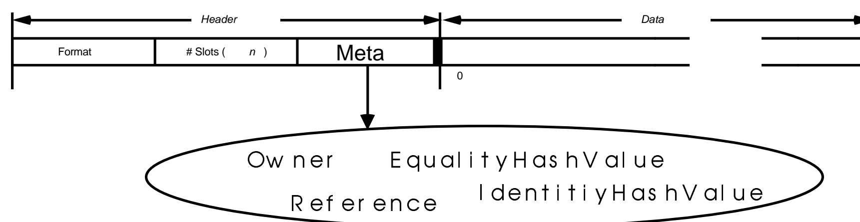


Abbildung 4.1: Der Meta-Slot

Meta-Daten-Zugriffsfunktionen

Ein Meta-Slot kann - wie andere Slots auch - über die Schnittstelle TSP mit den Operationen *tsp_getMeta* und *tsp_setMeta* manipuliert werden. Die Schnittstelle *tmmeta* bietet zusätzlich acht Operationen, die einzelne Meta-Werte, d.h. EqualityHashValue, IdentityHashValue, Referenz und Owner, lesen und schreiben.

Global eindeutige Objektrepräsentation

Ubiquitäre Ressourcen werden durch global eindeutige symbolische Referenzen repräsentiert. Eine Symbolische Referenz muß die einzige Repräsentative für eine ubiquitäre Ressource im Tycoon-System sein. Die Generierung von eindeutigen Objektidentifikatoren kann nicht garantiert werden. Zusätzlich dürfen die Symbole

keine komplexe Struktur besitzen, da komplexe Symbole genau so hohe Kommunikations-kosten verursachen können wie die Objekte selbst. Die Schnittstelle *Atom.ti* in Tycoons *stdenv* Bibliothek bietet zwei Funktionen für diesen Zweck:

```
1) atom.fromString(name :String) :Atom.T
```

Die Funktion *atom.fromString* generiert auf Basis eines vom Benutzer ausgewählten Strings ein unveränderliches Symbol. In unserem Prototyp wird während der automatischen Registrierung der Module für das dynamische Rebinden deren Namen als Eingabestring benutzt. In Tycoon-Bibliothek werden die Module eindeutig benannt, um Namenkonflikte zu vermeiden. Daher ist diese Funktion für die Generierung der global eindeutigen Objektidentifikator für unveränderliche Ressourcen wie Bibliotheksmodule geeignet. Für veränderliche Ressourcen sind zusätzliche dynamische Typprüfungs-mechnismen während des Rebindens des Moduls am Zielort notwendig, um sicher zustellen, daß der Typ der Ressource an der Quelle mit dem Typ des Moduls am Zielort übereinstimmt.

```
2) atom.unique(V :Ok value:V) :Atom.T
```

Die Funktion *atom.unique* generiert für das Tycoon-Objekt *value* ein Symbol, das sich aus Plattformidentifikation, Geräte-Identifikation und Zeitstempel zusammensetzt. Da der Compiler bei der Zuordnung dieser global eindeutigen Objektidentifikator (UID) für jedes Tycoon-Objekt beteiligt ist, kann diese UID als ein typsicherer Identifikations-mechanismus für Objekte in verschiedenen Objektspeichern, die von demselben Compiler stammen, benutzt werden.

Verwalten der symbolischen Referenzen mit Hilfe der Hashtabelle

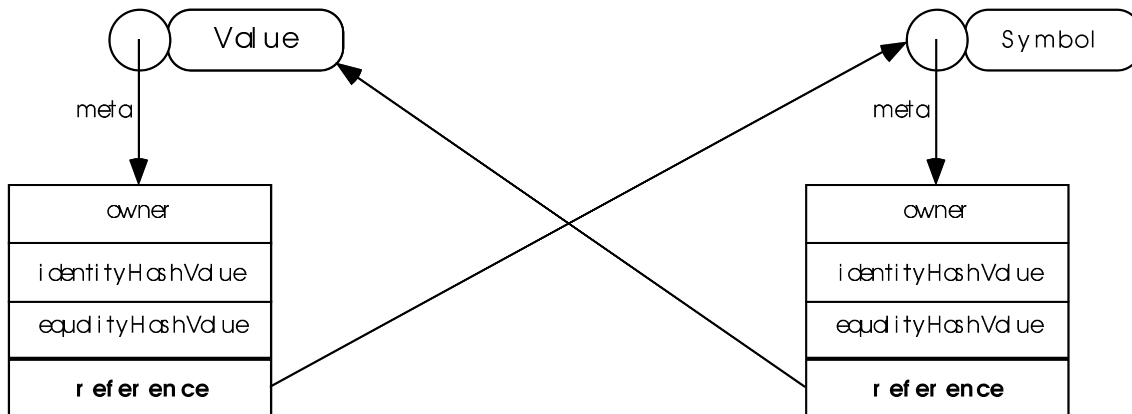
Für die Verwaltung der symbolischen Referenzen ist in jedem Objektspeicher eine persistente Hashtabelle vorgesehen. Die generierten symbolischen Referenzen werden in diese Tabelle eingetragen. Das Eintragen der Objekte in die Hashtabelle und deren Suche geschieht mittels ihres Hashwertes. Um das über das Netz ankommende Symbol in der lokalen Hashtabelle identifizieren zu können, ist es erforderlich, daß das gleiche lokale Symbol denselben Hashwert besitzt. Zu diesem Zweck ist die Hashfunktion *hash.EqualityhashValue* implementiert worden.
Diese generische Funktion akzeptiert beliebige Tycoon-Objekte als Eingabe und generiert in Abhängigkeit vom Inhalt des Objektes einen Integerwert als Hashwert. Dieser Wert wird im Meta-Slot des Objektes eingetragen und wird beim Registrieren und Aussuchen von Symbolen in Hashtabellen benutzt.

Zuordnung vom Objekt zum Symbol und umgekehrt

Das Etablierung einer Zuordnung zwischen Tycoon-Objekt und dessen UID wird durch Aufruf der Funktion:

```
atom.connect(symbol :Tagged value :Tagged) :OK
```

ermöglicht. Dabei erhält der Meta-Slot vom Objekt eine Referenz auf UID und umgekehrt. Diese Zuordnung ist in der Abbildung (4.2) dargestellt.



Abildung 4.2: Zuordnung vom Objekt zum Symbol

Markierung eines Objektes als ubiquitäre Ressource

Ein Wert vom Typ *tsp_Format* wird zur Defintion des Formates der Struktur eines persistenten Objektes im Objektspeicher benutzt (siehe Kapitel 3). Dieser Wert, der sich im Objektkopf befindet, kann mit in TSP vorgesehenen Operationen abgefragt und modifiziert werden. Für die Markierung eines Objektes als ubiquitäre Ressource wird ein Bit des Formatwertes zur Markierung benutzt. Anhand einer Abfrage des Objektformates kann festgestellt werden, ob es sich um eine ubiquitäre Ressource handelt.
Die Markierung wird wie folgt durchgeführt:

```
define tmstore-Format-FLAG-UBIQUITOUS  (0x40000000L)

tsp_setFormat(STORE, oid, tsp_getFormat(STORE, oid) |
                    tmstore-Format-FLAG- UBIQUITOUS)
```

Eine Beispielapplikation

Sei *margot* ist ein beliebiges Tycoon-Objekt:

```
let margot = tuple let name ="Margot" let age =35 end
```

Eine global eindeutige symbolische Referenz worde von der Funktion *atom.fromString* generiert:

```
let symbol = atom.fromString("nett")
```

Im Zuge der Ausführung dieser Funktion wird der Hashwert für *symbol* berechnet und *symbol* in die persistente Hashtabelle eingetragen.
Die Zuordnung zwischen den Tycoon-Objekt *margot* und dem Referenz *symbol* geschieht durch den Aufruf der Funktion *atom.connect*:

```
atom.connect(symbol margot)
```

Anschließend wird das Objekt *margot* als ubiquitäre Ressource markiert.

```
let format = store.getFormat(margot) |
        tmstore_FORMAT_UBIQUITOUS

store.setFormat(margot  format))
```

Garbage Collection von Symbolen

Aufgabe der Speicherfreigabeoperation (garbage collection) [Math92] im Tycoon-System ist die Freigabe des durch nicht mehr erreichbare Objekte belegten Speichers. Dabei gilt:

- Das persistente Wurzelobjekt des Speichers ist erreichbar.
- Jedes fixierte Objekt ist erreichbar.
- Jedes Objekt, dessen Referenz in einer lokalen Zustandsvariablen eines TML-Evaluators gehalten wird, ist erreichbar.
- Ist ein Objekt *O* mit dem Format value-Array-Format oder Closure-Format erreichbar, so sind alle Objekte, die über Werte in O referenziert werden, ebenfalls erreichbar.
- Keine anderen Objekte sind erreichbar.

Wie bereits diskutiert wurde, werden die Symbole in einer lokalen persistenten Hashtabelle registriert. Um die unbenutzten Symbole für die Garbage-Collection auffindbar zu machen, werden alle Referenzen der Tabelle auf Symbole als *Weak-Reference* [Nel91] definiert.

Wir erweitern die oben genannten Bedingungen für die Erreichbarkeit eines Objektes:

- Ein Objekt ist erreichbar,wenn mindestens eine erreichbare Referenz darauf zeigt.
- Falls die einzige Referenz, die auf das Objekt zeigt, eine Weak-Reference ist und die Garbage-Collection aktiv wird, soll eine *cleanup-function*, die mit dem Objekt implementiert ist, aufgerufen werden, um die Referenz für ungültig zu erklären.

Dieselbe Funktion wird für das Löschen von Symbolen benutzt. In der Hashtabelle ist die *cleanup-function* durch eine *delete* Funktion implementiert:

```
let cleanup(X<:Ok  weak :weakReference.T(X)  hart :X) =
            hashTable.delete(table weak)
```

4.2.2 Integration des dynamischen Rebinden in den Linearisierungsalgorithmus

Modulare Integration des dynamischen Rebindens in den im TSP vorhandenen *extern-* bzw. *intern-* Algorithmus, die im Abschnitt 4.1 ausführlich diskutiert werden, umfaßt Implementierung der Funktionen *writeHandler* und *readHandler* für dynamisches Rebinden.

Realisierung der *WriteHandler* für dynamisches Rebinden

Die Aufgaben der *writeHandler*-Funktion für dynamisches Rebinden sind im wesentlichen:

- Identifizierung der ubiquitären Ressourcen im Objektgraph.
- Auffinden der symbolischen Referenzen.
- Linearisieren der symbolischen Referenzen anstelle von ubiquitäre Ressourcen in den Datenstrom.

```
tsp_OID writeHandler(tsp_Store store, tsp_XDR xdr, tsp_OID oid,
                     void *handle, tsp_WriteFunction write, void *data);
```

Der *WriteHandler* fragt mit Hilfe der Funktion *tsp_getFormat* das Format von *oid* ab. Anhand des Formats kann festgestellt werden, ob *oid* ein ubiquitäres Objekt ist. Wenn dies nicht der Fall ist, wird *tsp_defaultWriteHandler* aufgerufen und das Objekt *oid* mit Deep-Copy-Operation linearisiert.
Ist das Objekt *oid* ubiquitär, wird über seinen Meta-Slot die dazugehörige symbolische Referenz gefunden. Der Meta-Slot einer ubiquitären Ressource beinhaltet eine Referenz auf ihre symbolische Referenz. Dies geschieht durch das Aufrufen der Operation:

```
tsp_oid tmmeta_getReference(oid)
```

Die Funktion *tmmeta_getReference(oid)* liefert eine symbolische Referenz von *oid* zurück.
Jetzt muß die symbolische Referenz von *oid* in den Datenstrom linearisiert werden. Um Versionsabhängigkeiten zu vermeiden und Portabilität zu gewährleisten, wird der Objektkopf symbolischer Referenz durch die im TSP für diesen Zweck vorgesehene Funktion *tsp_externAttributes* behandelt:

```
tsp_Word  tsp_externAttributes(tsp_Store store, tsp_XDR cdr, tsp_Tagged taggedMeta,
                               tsp_Format format, tsp_Bool fImmutable,
                               tsp_Word nSlots, void *handle,)
```

Abschließend wird *tsp_defaultWriteHandler* (s. Abschnitt 4.1, Deep-Copying) für die Linearisierung der symbolischen Referenz aufgerufen.

Realisierung der *readHandler* für dynamisches Rebinden

Diese Funktion wird vom *intern*-Algorithmus aufgerufen und soll die folgenden Aufgaben erfüllen:

- Identifizieren der symbolischen Referenz im über das Netz ankommenden Datenstrom.
- Aufsuchen der gleichen lokalen, symbolischen Referenz in der persistenten Hashtabelle.
- Lokalisieren der entsprechenden lokalen ubiquitären Ressource im Objektspeicher.
- Integration der lokalen Ressource in Objektgraphen.

```
tsp_OID  readHandler(tsp_Store store, tsp_XDR xdr, tsp_Tagged taggedMeta,
                     tsp_Format format, tsp_Bool fImmutable,
                     tsp_Word nSlots, void *handle, tsp_ReadFunction read);
```

Während der Linearisierung der Tycoon-Objekte werden zuerst die Informationen, die sich im Objektkopf befinden, linearisiert und über das Netz geschickt. Dementsprechend wird in den Delinearisierungsverfahren zuerst diese Informationen empfangen. Diese erlaubt dem *intern*-Algorithmus die ankommenden Daten entsprechend in den Objekspeicher zu integrieren.
Der *ReadHandler* für dynamisches Rebinden stellt mittels der Formatinformation fest, ob zunächst eine symbolische Referenz erwartet werden soll. Wenn dies nicht der Fall ist, wird *tsp_defaultReadHandler* aufgerufen.
Wenn anhand der Formatinformation eine symbolische Referenz identifiziert wird, wird weiter der Hashwert des Symbols, der sich ebenfalls im Objektkopf befindet, gelesen. Dieser Wert wird zum Aufsuchen der gleichen lokalen Referenz in der lokalen Hashtabelle benutzt. Bei Erfolg wird eine entsprechende lokale ubiquitäre Ressource über die lokale Referenz lokalisiert und in das Objektgraphen integriert.

Wenn das ankommende Symbol in der lokalen Tabelle nicht identifiziert werden, bietet *ReadHandler* zwei Möglichkeiten:

- Nach dem Auslösen einer Fehlermeldung wird die *intern*-Operation unterbrochen.
- Die automatische Replikation wird ausgelöst und das Verfahren wird fortgesetzt. Dieser Fall hat die Aufforderung der vermißten Ressourcen vom Sender der Daten und anschließender Vervollständigung des Objektgraphen auf der Empfängerseite zur Folge. In Abschnitt 4.3 wird dieser Fall ausführlich diskutiert.

4.2.3 Verwendung des dynamischen Rebinden in verteilten Anwendungen

Alle Tätigkeiten im Tycoon System (Übersetzen, Binden, Ausführen, Fehleranalyse, Definition von Systemparametern) werden durch Kommandos (top level phrases) ausgelöst.
Für die automatische Registrierung von Modulen für dynamisches Rebinden ist ein *do* Kommando definiert, das wie ein Schalter funktioniert:

```
do set dynamicLinking true;
```

Nach der Ausführung dieses Kommandos wird ein Modul im Verlauf des Imports als ubiquitäre Ressource registriert.

```
import window;

import basicServer;
```

Diese hat eine grob granulierte Registrierung der Module zur Folge. Das bedeutet ,daß die exportierten Funktionen explizit als ubiquitäre Ressourcen registriert werden müssen.

Zusätzlich sind Funktionen für das manuelle Registrieren beliebiger Tycoon-Sprachobjekte für dynamisches Rebinden vorgesehen. Die Schnittstelle *Atom.ti* stellt die Operation *atom.register* für die Registrierung eines Objektes als ubiquitäre Ressource zur Verfügung. In folgenden werden das Modul *Print*, die Funktion *print.string* und das Tuple *address1* für dynamisches Rebinden als ubiquitäre Ressourcen registriert.

```
atom.register(print "print")
atom.register(print.string  "p.string")
let address1 = tuple .......end
atom.register(address1 "address1")
```

4.3 Implementierung der automatischen Replikation

Dynamisches Rebinden der ubiquitären Ressourcen kann nicht, wie es im Abschnitt 4.2 beschrieben ist, erfolgreich ausgeführt werden, wenn die symbolischen Referenzen im Zielort nicht identifiziert werden. Das bedeutet, es sind keine lokale ubiquitäre Ressourcen für die ankommenden Symbole vorhanden oder sie sind nicht für dynamisches Rebinden registriert. Dieser Fall wird durch das Auslösen einer Fehlermeldung aufgefangen, was Mißerfolg der Datenübertagung zur Folge hat.

Aber bei der Existenz bidirektionaler Kommunikationskanäle kann der Fall durch die Anforderung der expliziten Übertragung von fehlenden Ressourcen gelöst werden. Die fehlende ubiquitäre Ressourcen können im Zielort gleich für dynamisches Rebinden registriert werden. Der Vorteil dieses Designs liegt darin, daß eine weitere Übertragung der gleichen Ressourcen mittels dynamischen Rebindens stattfinden kann. Diese Lösung, die eine Kombination der beiden Methoden Deep-Copying und dynamisches

Rebinden ist, führte zur Entwicklung der Methode automatische Replikation, deren Konzept in Kapitel 2 ausführlich dargestellt ist.

Die Aufgabe der automatischen Replikation besteht im wesentlichen aus:

- Vorläufiges Eintragen der nicht identifizierbaren Symbolen in den Objektgraph im Zielobjektspeicher.
- Verwaltung der unbekannten Symbolen.
- Traversieren des Objekgraphen und Ersetzen der Symbolen durch deren Ressourcen, die vom Sender mit Deep-Copying verschickt worden sind.
- Registrieren dieser Ressourcen als ubiquitäre Ressourcen im Zielobjektspeicher.

4.3.1 Integration der automatischen Replikation in den Linearisierungsalgorithmus

Automatische Replikation greift den *intern*-Algorithmus während der Delinearisierung der symolischen Referenzen und bei der Traversierung des Objektgraphen zum Abschluß der Importierung der Daten an. Realisierung von *ReadHandler* und *SlotHandler* für automatische Replikation und deren Aufruf von *tsp-intern* realisiert die modulare Integration der automatischen Replikation in den TSP vorhandenen Linearisierungsalgorithmus.

Anpassung der *readHandler* für automatische Replikation

Um die nicht identifizierte Symbole in einer späteren Phase durch nachträglich übertragene Objekte ersetzen zu können, werden diese vorläufig in den Objektgraphen eingetragen. Diese wird durch die Anpassung der *readHandler* für automatische Replikation realisiert.
Der auf diese Weise entstehende Objektgraph im Objektspeicher kann allerdings noch nicht von anderen Objekten referenziert und benutzt werden. Es bedeutet, daß die Übertragung der Daten noch nicht vollständig ist. Es geschieht zuerst in einer späteren Phase, in der die Symbole im Objektgraph durch die entsprechende Ressourcen ersatz werden.

Anpassung der *slotHandler* für automatische Replikation

Die Erkennung und Behandlung der unbekannten symbolischen Referenzen, die sich im Objektgraphen befinden, ist die Aufgabe der *slotHandler*. Diese Funktion wird einmal pro Slot in jedem Vektorobjekt des Objektgraphen aufgerufen. Mit Hilfe dieser Funktion kann der Benutzer ein beliebiges Verfahren auf einzelner Objekte des von der *intern* Funktion neu rekonsturierten Objektgraphen ausführen.
Es wird geprüft, ob der im Objektgraphen besuchte Slot einen symbolischen Referenzen enthält. Die Symbole (*oidChild*) und ihre Adressen (*oidParen, iIndexParent*) im Objektgraphen werden in eine Hashtabelle, die für die Verwaltung der unbekannten

Symbole initialisiert und für die Dauer der Datenübertragung persistenz vorhanden ist, eingetragen. Dies geschieht durch Aufruf der Funktion *tlsymlock_insert*:

```
tlsymlock_insert(oidChild, oidParent, iIndexParent)
```

Der Benutzer der automatischen Replikation kann beim Aufrufen der Funktion *tlsymlock_packSymbols* die unbekannten Symbole zurückschicken und die Ressourcen dafür anfordern. Beim Ankommen der originalen Ressourcen werden diese mit Hilfe der Hashtabelle an die richtige Stelle in den Objektgraph integriert und gleichzeitig als ubiquitäre Ressourcen im Zieladreßraum registriert. Diese geschieht mit

```
void tlsymlock_updateLocations(tsp_OID *oidValues)
```

4.3.2 Verwendung der automatische Replikation in verteilten Umgebungen

Die *intern* Funktion im Tycoon-System ist in zahlreichen Anwendungen wie z.B. Datenaustausch zwischen Tycoons Objektspeichern, kompakte Speicherung verschiedener Datenversionen, Backups und Datenbank-Logging, die Erzeugung eigenständiger Tycoon-Programme in sogenannten Boot-Dateien, usw. involviert. Das Auslösen der automatischen Replikation in *intern* setzt die Existenz bidirektionaler Kommunikatioinskanäle voraus. Diese Bedingung wird nicht von allen oben aufgezählten Anwendungen erfüllt. Deshalb ist ein Mechanismus für das Ein- bzw. Ausschalten der automatischen Replikation durch den Anwender vorgesehen.

```
do set automatischeReplikation true;
```

Daraufhin wird in der *intern* Funktion die automatische Replikation ausgelöst.

4.3.3 Integration der automatischen Replikation in Tycoons polymorphe Client-Server-Programmierumgebung

Polymorphe persistente Client/Server-Programmierung [MG96] in Tycoon erfordet den Austausch der Programmcodes zwischen Client und Server, was den Einsatz der Bindungstechniken während der Datenübertragung notwendig macht.

Die Tycoon-Bibliothek *csenv* beinhaltet die generischen Basismodule für RPC im Tycoon -System. Diese Modulen unterstützen die RPCs zwischen getrennten Tycoon-Prozessen, die auf verschiedenen Objektspeichern laufen. Die generische Schnittstelle *value.ti* bietet die Operationen zum Austausch der Daten als Paket zwischen Client und Server.

Die Integration der automatischen Replikation erfolgte im Datenaustauschprotokoll, das durch die Module *value.receiveWithProtocol* und *value.sendWithProtocol* realisiert wird. Ein Paket kann *Daten*, *Symbol* (unbekannte Symbole), *additionalValues* (fehlende

Daten) oder *acknowledge* (Bestätigung für die Vollständigkeit der zu transferierenden Daten) beinhalten. Die Funktion *receiveWithProtocol* liest das ankommende Paket und behandelt sie entsprechend seinem Inhalt. Beim Eintreffen der unbekannten Symbole wird die automatische Replikation ausgelöst, d.h. die Symbole werden zuerst in den Objektgraphen integriert und gleich in der Hashtabelle registriert. Der Inhalt der Hashtabelle für nicht identifizierte Symbole wird als *Symbol*-Paket mit Aufruf der Funktion *sendWithProtocol* zurückgeschickt und es wird auf das Ankommen eines Paketes mit *additionalValues* (die eigentlichen Codes) gewartet.
Der Empfänger des *Symbol*-Paketes sendet die entsprechenden lokalen Ressourcen dieser Symbole als *additionalValue*-Paket zurück und wartet auf ein *acknowledge*-Paket. Die ankommenden *additionalValues* werden an der Stelle der entsprechenden Symbole in den Objektgraphen integriert. Dies geschieht mit einem Aufruf von *upDateLocations*:

```
symloc.upDateLocations(p.Values)
```

Wird kein neues, unidentifiziertes Symbol gemeldet, ist die Datenübertragung vervollständigt, und das Datentransfer wird mit Übersenden des *acknowledge*-Paketes zur Bestätigung der ankommenden Daten abgeschlossen.

```
writePacket(sock acknowledge)
```

In der Abbildung (4.3) ist der Protokollablauf für den Austausch der Datenpakete dargestellt.

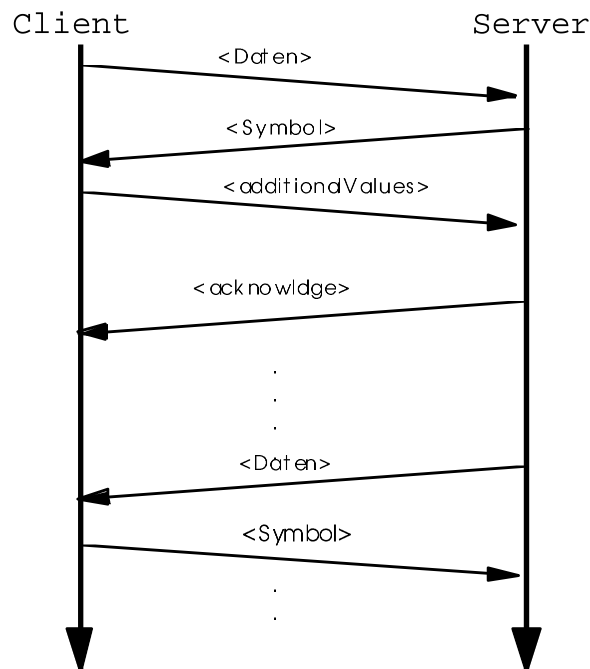


Abbildung 4.3: Protokollablauf des Paketaustausches

Kapitel 5

Performanzsteigerung

In Tycoon können verteilte Anwendungen mit Hilfe migrierender Threads oder polymorpher entfernter Prozeduraufrufe (höherer Ordnung) realisiert werden [MMS95b]. Im folgenden werden die Wirkungen des dynamischen Rebindens im Zusammenhang dieser Konzepte mittels Beispielapplikationen verdeutlicht.

5.1 Wirkung des dynamischen Rebindens bei der Migration von Threads

Dynamisches Rebinden ist für migrierende Threads unentbehrlich. Migration von Threads ist das Hauptkonzept der verteilten Programmierung im Tycoon-System. Die Datenobjekte, die Threads enthalten, können eine Vielzahl von Bezügen (Bindungen) auf ubiquitäre Ressourcen haben, die meistens große Strukturen besitzen, wie z.B. Implementierung der Massendaten. Den migrierenden Threads ermöglicht die Rebindtechnik die Bindung an ubiquitäre Ressourcen, die in den von Threads besuchten Knoten vorhanden sind.
Das folgende Thread wurde in einer heterogenen Umgebung zwischen Tycoon-Systemen, migrieriret:

```
import ...
  cCallback chekpoint iter thread volatile..
  brush button pen color window menu outputDevice...

let activity(self :thread.T(ok)) =
  begin
    let var width = 100
    let var x = 0

    let paint(...) = ....color ... brush ...
    let m = menu.newPopUp(...)
    installCallbacks(.... w ... m ... paint ...)
    loop
```

```
      outputDevice.drawLine(x ...)
       x := {x+1} % width
     end
   end

let t = thread.new(activity)
....
dynamic.extern(t ....)
```

Das Thread *t* beinhaltet Referenzen auf das Modul *Starview*, eines der größten Module, im Tycoon-System. Die Größe des transitiven Funktionsabschlusses ist 6,5 MB, daß heißt beim Migrieren muß 6,5 MB Code gesendet werden.
Diese Menge reduziert sich auf 60 KB, wenn Standardmodulen, wie *cCallback, chekpoint, iter, thread, volatile,.. brush, button, pen, color, window, menu, outputDevice*, dem dynamischen Rebinden unterliegen. Aus diesem Grund ist dynamisches Rebinden für die Migration von Threads, besonders in WANs, sehr entscheidend.

5.2 Automatische Replikation in Client-Server-Anwendungen

Im folgenden Beispiel wird die Wirkung der automatischen Replikation in Bezug auf Performanzsteigerung von Client-Server-Anwendungen im Tycoon demonstriert:

Der Server *searchOp* exportiert die polymorphe Function von höherer Ordnung *search*. Die Funktion *match* kann als Eingabeparameter von Aufruf zu Aufruf variieren. Das Rückgabeargument, die Funktion *image.display(:X)*, besitzt einen sehr großen Funktionsabschluß und wird beim Aufruf der Funktion zur Seite des Clients geschickt, um auf dem Bildschirm die relevante Ausgabe zu erzeugen.

```
Let SearchOp = Tuple
  search(A <:Ok  Element :A   match :Fun(e:A k:A) :Bool) :Fun()
:Ok
.....
end

let searchOp :SearchOp = tuple
  let search(A <:Ok  Element :A  match :Fun(e:A k:A) :Bool)
             :Fun():Ok =
  begin
   .....
   let f() :Ok = image.diplay (Element)
   f
  end
end

let registerService = rpcServer.register(server "searchOp"
                                    searchOp)
```

Ein entfernter Aufruf dieser Funktion kann wie folgt sein:

```
let match(A <:Ok  e:A  k:A) :Bool = begin
......
end

let d = searchOp.search(Meyer  match)

d()
```

Mehrfaches Aufrufen dieser Funktion hat das wiederholte Übertragen der Funktion *image.display* mit dem gesamten Funktionsabschluß und ihre mehrmalige Installierung auf der Clientseite zur Folge, was

a) zeitkritisch für die Netzwerkverbindung und
b) platzkritisch für den Client ist.

Die automatische Replikation der Funktion *image.display* hat das Registrieren der Funktion als ubiquitäre Ressource auf der Clientseite beim ersten Aufruf zur Folge. Bei einem weiterern Aufruf der Funktion *searchOp.search* wird der Code der Funktion *image.display* nicht mehr gesendet, sondern die an der Clientseite automatisch installierte Funktion *image.display* in das vom Server ankommende Ergebnispaket eingebunden. Dies bedeutet, daß die Funktion *image.display* **nur einmal** über das Netz geschickt und nur **einmal** im Client-Objektspeicher installiert wird.

Kapitel 6

Zusammenfassung und Ausblick

Die in dieser Arbeit entwickelten flexiblen Bindungstechniken für ubiquitäre Ressourcen im Tycoon-System reduzieren den Netzwerksverkehr für verteilte Anwendungen drastisch. Dadurch wird ein problemloser Einsatz nichttrivialer aktivitätsorientierter Anwendungen im Internet erreicht.

Verteilte Tycoon-Anwendungen implizieren den Austausch von persistenten Daten, Codes und Threads. Es hat sich gezeigt, daß transitive referentielle Abschlüsse von Funktionen und Modulen sehr schnell wachsen können, was ihren Versand im Internet kritisch macht. Ubiquitäre Ressourcen (Bibliotheks-Module, Datenbanken, etc.) stellen einen wesentlichen Anteil an den zu übertragenden Daten. Durch das dynamische Rebinden der ubiquitären Ressourcen wird die Übertragung dieser Daten ohne Netzbelastung erreicht. Außerdem bleiben die Lokalität von Programmen und die Autonomie migrierender Threads erhalten.

Die Sprachobjekte im Tycoon-System befinden sich in einem Objektspeicher und können unter Benutzung des Tycoon-Store-Protocol modifiziert und linearisiert werden. Deswegen hat das TSP sich als Ausgangspunkt der Implementierungsphase angeboten. Durch die Erweiterung der Objektstruktur in TSP wurde die Identifizierung und Verwaltung ubiquitärer Ressourcen und derer symbolischen Referenzen ermöglicht. Diese Erweiterung wird außerdem in anderen Zusammenhängen, wie z.B. bei der Autorisierung von Datenobjekten im Objektspeicher auch benutzt.
Die Unterscheidungsmöglichkeit der ubiquitären Ressourcen ist durch die Einführung eines entsprechenden Formattyps vorgesehen. Die symbolischen Referenzen sind global eindeutig und besitzen eine sehr einfache Struktur. Sie werden in Hashtabellen verwaltet, was einfaches Aufspüren im Sender-Objektspeicher bzw. einfache Identifizierung im Ziel-Objektspeicher erlaubt.

Ein zentrales Problemfeld der Arbeit ist der vorgegebene Datenlinearisierungs-algorithmus, welcher sehr spezifisch für Deep-Copying gedacht ist. Hier wird eine

strenge Modularisierung vorgenommen. Dadurch wird ein besonders höher Grad der Flexibilität für die Integration neuer Bindungstechniken erreicht.
Bei Auslösung der automatischen Replikation bei der Nichtidentifizierung von symbolischen Referenzen werden die fehlenden Ressourcen nachgesendet. Diese werden anschließend im Zielknoten als ubiquitär registriert. Jede folgende Übertragung der gleichen Ressourcen unterliegt daraufhin dem dynamischen Rebinden. Automatische Replikation ist bereits in Tycoons Client-Server-Programmierumgebung [MG96] integriert.

Die genannte Bindungsmechanismen sind keine allgemeine Alternative zum Deep-Copying. Sie sind nur unter Erfüllung der für sie notwendigen Voraussetzungen, d.h. der Existenz ubiquitärer Ressourcen für dynamisches Rebinden und bidirektionaler Kommunikationskanäle für automatische Replikation, einsetzbar.
Es sind Schaltmechanismen als Kommandos realisiert. Der Benutzer besitzt die Kontrolle über das automatische Registrieren von Modulen als ubiquitäre Ressourcen und die Aktivierung der automatischen Replikation. Zusätzlich sind Funktionen für das manuelle Registrieren beliebiger Tycoon-Sprachobjekte für dynamisches Rebinden vorhanden.

Ausblick

Die Primitive für dynamisches Rebinden und automatische Replikation sind auf Grund ihrer persistenten, polymorphen Implementierung hochgeradig wiederverwendbar. Die modulare Integration der Bindungstechniken in Tycoons Linearisierungsalgorithmus bietet gute Möglichkeiten der Weiterentwicklung der einzelnen Teile der Implementierung. Zum Beispiel können Netzwerk-Referenzen analog zu symbolischen Referenzen realisiert und in den Linearisierungsalgorithmus integriert werden. Die Realisierung externer Representationen lokaler Referenzen durch Netzwerk-Referenzen führt zu einer vereinfachten Verwendung immobiler, insbesondere veränderlicher, Ressourcen in aktivitätsorientierten verteilten Anwendungen.
Auf diese Weise kann ein Rahmen zur Unterstützung effizienter Mobilität aller referenzierten Ressourcen in einer verteilten Tycoons Anwendung bereitgestellt werden.

Es ist denkbar, eine Instanz einzuführen, die relative Kosten der Mobilität, Lokalität und entfernter Aufrufe auf Basis von Nachbarschaftsinformationen pro Netzwerkverbindung berechnet. Auf dieser Basis könnten in verteilten Aktivitäten beteiligter Ressourcen während der Linarisierung dynamisch behandelt werden. Durch die dynamische Koordination der existierenden Bindungs- und Linearisierungstechniken kann dann eine optimierte Übertragung komplex zusammengestellter Daten, wie z.B. Threads, in WANs erreicht werden.

Literaturverzeichnis

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming Languages,* volume 242 of Lecture Notes *in Computer Science*. Springer-Verlag, 1986.

[Car89] L. Cardelli. *Typeful programming*. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, May 1989.

[Car94] L. Cardelli. Obliq: *A language with distributed scope*. Technical report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, June 1994.

[Corb91] J.R. Corbin. *The Art of Distributed Applications*. Sun Technical Reference Library, Springer-Verlag 1991.

[Jul88] E. Jul, *Object Mobility in a Distributed Object-Orinted System*. PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1988.

[Jul89] E. Jul. *Migration of Light-weight Processes in Emerald*. Operation Systems Technical Comittee Newsletter, 3(1), pages 25-30, 1989.

[Math92] B. Mathiske. *Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung*. Diplomarbeit, Fachbereich Informatik, Univeristät Hamburg, 1992.

[Math96] B. Mathiske. *Mobilität in persistenten Objektsystemen*, PhD thesis Fachbereich Informatik, Universität Hamburg, 1996.

[Matt93] F. Matthes. *Persistente Objektsysteme*: Integrierte Datenbankentwicklung und Programmerstellung. Springer-Verlag, 1993.

[MG96] M. Göllnitz. *Polymorphe, Persistente Client/Server-Programmierung mit dynamischer, hierarchischer Adreß-Auflösung,* Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1996.

[MMM93] B. Mathiske, F. Matthes, and S. Müßig. *The Tycoon system and library manual*. DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, 1993.

[MMS95a] B. Mathiske, F. Matthes, and J.W. Schmidt. *On Migrating Threads*. In Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel, June 1995.

[MMS95b] B. Mathiske, F. Matthes, and J. W. Schmidt. *Scaling Database Languages to Higher-Order Distributed Programming*. In Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy. Springer-Verlag, September 1995.

[MMS95c] F. Matthes, R. Müller, and J. W. Schmidt. *Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol*. To appear in Fully Integrated Data Environments, Springer-Verlag, 1995.

[Mun93] D.S. Munro, *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, Departement of Mathematical and Computational Sciences, University of St. Andrews, Scotland, 1993.

[Nel93] G. Nelson, Hrsg. *Systems programming with Modula-3*. Series in innovative technology prentic Hall, Englewood Cliffs, New Jersey, 1991.

[OSF93] OSF. OSF DCE Administration Guide - Core Components. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[Schi93] A. Schill, DCE- Das OSF Distributed Computing Environment, Springer-Verlag 1993.

[SUN90] Sun Microsystems. SUNOS 4.2. Network Programming Guide. Sun Microsystems, inc. 1990.

[Wai89] F. Wai, Distributed PS-algol. In R. Rosenber and D.Koch, editors, In *Proceedings of the 3rd International Workshop on Persistent Object Store Systems, Newcastle*, NSW, pages 126-140. Springer-Verlag, January 1989.